

Composition, Cooperation, and Coordination of Computational Systems

Johannes Reich, johannes.reich@sap.com

Draft version 2016-02-23

Abstract. A system model is developed where the criterion to partition the world into a system and a rest is based on the functional relation between its states. This approach implies that the gestalt of systems becomes very dynamic. Especially interactions between systems may create temporary super systems "on the fly". The reference to the function notion establishes the link to computation. It is shown that being computable is a system property which is compositional in an abstract mathematical sense but is, because of recursion, emergent from a computational point of view.

Based on the distinction between the intended determinism versus non-determinism of the interaction, two classes of system interactions are distinguished: composition and cooperation. Composition means that by interaction super systems are created and the interacting systems become subsystems. Cooperation means that systems sensibly interact loosely without creating any identifiable super system function and therefore without super system creation from the perspective of the interacting systems.

Cooperative system interactions are described with the help of the protocol notion based on shared states, interpreted as the stateful exchange of characters via Shannon channels between roles - the projection of systems onto the interaction channels. It is shown that roles can be internally coordinated by a set of rules, leading to a very flexible process notion which unfolds its full potential only in a new type of execution environment capable of executing spontaneous transitions.

The system model has immediate implications for componentization which can be viewed as an attempt to either hide functional recursion and provide only functionality whose composition behavior is easy to understand, or to provide loosely coupled interactions via protocols. To be complete, component models should therefore at least support three general classes of components: one for loose coupling, one for hierarchical composition, and one for pipes.

1 Introduction

Civil, mechanical, electrical, or software engineering - they all deal with the design of systems. The term "system engineering" was coined in the Bell Laboratories in the 1940s (e.g. [13]). According to Kenneth J. Schlager [51], a main driver of the field was the complexity of the composition behavior of components.

Often seemingly satisfactory working components did not result in satisfactorily working systems. The focus on systems was transferred to other disciplines like biology (e.g. [8]) or sociology (e.g. [42])

Especially with the advent of cyber physical systems (e.g. [32, 1]), also the software engineering discipline articulates the importance of a unifying view (e.g. [55]) as the different disciplines complement each other more and more.

As a base for such a unifying view, I propose a common system notion which is grounded on the notion of a system's functionality, a notion that is tightly related to the concepts of computability and determinism of computer science.

Although systems can be described as isolated entities, their main purpose is to interact. And as will be shown in this article, due to their interactions at least two very different system relations arise: composition and cooperation.

System composition means that by their interaction super systems are created and the interacting systems become subsystems. As the system notion is based on the function notion, the concept of system composition can be traced back to the concept of composition of functions which is at the heart of computability. Especially for software components, the borrowed distinction between non-recursive and recursive system composition becomes important. As is illustrated in Fig. 1, system composition leads to hierarchies of systems which are determined by the 'consists of'-relation between supersystems and their subsystems.

Hierarchical system composition

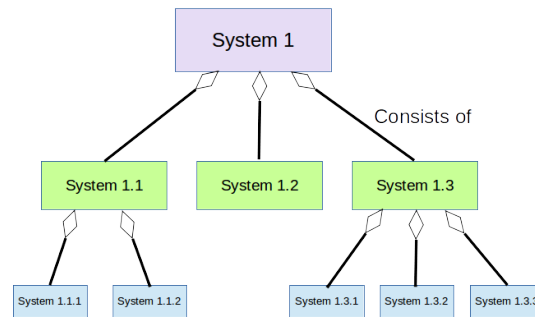


Fig. 1. An example of a hierarchically composed system, namely system 1. It consists of 3 subsystems which by themselves consist of a couple of further subsubsystems.

Quite contrasting, system cooperation means that systems interact sensibly somehow "loosely" without such super system creation. In essence, this means that no super system function can be identified as the interaction remains non-deterministic. This is typically the case in interaction networks. As an example, Fig. 2 sketches a cutout of a network of business relations between a buyer, a

seller, its stock, a post and a bank. In these networks none of the participants is in total control of all the other participants: the participants' interactions don't, in general, determine the participants' actions. These networks are open in the sense that we will never be able to describe them completely.

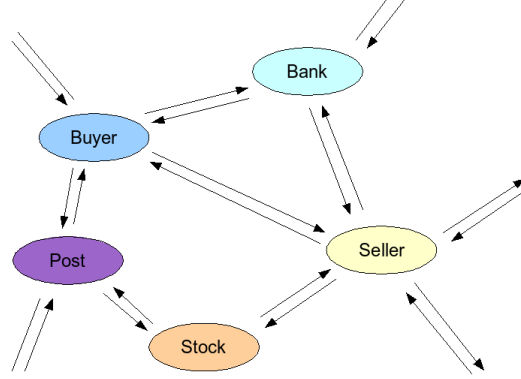


Fig. 2. A cutout of an open business network.

The distinction between system composition and cooperation suggests a component model which distinguishes between compositional and cooperative components. The relevance of system composition for component oriented development has already been pointed out for example by Kung-Kiu Lau and Zheng Wang [31]. The desire to use compositional components as building blocks with a strict hierarchical composition behavior leads to the requirement that compositional components represent a complexity border hiding all functionality based on recursion.

However, the relevance of system cooperation for component models seems to be less well understood, although the relevant entity to describe these interactions is well known since the 1960s: protocols (e.g. [50, 58, 25]). Protocols describe nondeterministic, asynchronous, and stateful interactions we find in interaction networks showing no formal hierarchy [47]. In their overview on component models Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis and Michel R. V. Chaudron [14] distinguish between "operation based" and "port based" interface support and show thereby that many currently important component models indeed do not support protocol declarations.

As demonstrated in this article, an important consequence of the recursive nature of computation is that the identification of discrete systems is not computable.

The structure of the article is as following. In section 2, a system definition is provided and the concept of computable functionality is recapitulated to give the whole discussion a sound base. In section 3, the issue of system composition is

investigated. A distinction is made between the operation of system composition and the property of being compositional for some system properties. Sequential, parallel and recursive Loop- and While-compositions are identified. In section 4, the cooperation of systems as the nondeterministic form of their interaction is examined. Cooperative system interactions are described with the help of the protocol notion based on shared states, interpreted as the stateful exchange of characters via Shannon channels between roles - the projection of systems onto the interaction channels. In section 5, the focus is shifted to the inner structure of cooperating systems which are named processes. It is shown that roles can be internally coordinated by a set of rules, leading to a very flexible process notion. Related work is presented in section 6. Here, I dwell on abstract data types, objects, reactive systems, and the system models of Manfred Broy and Rajeev Alur as well as the BIP component framework of Joseph Sifakis et al. In the final section 7, the concepts and results of this article are discussed with respect to their possible implications for software engineering and componentization as well as for other areas of science.

1.1 Preliminaries

Throughout this article, elements and functions are denoted by small letters, sets and relations by large letters and mathematical structures by large calligraphic letters. The components of a structure may be denoted by the structure's symbol or, in case of enumerated structures, index as subscript. The subscript is dropped if it is clear to which structure a component belongs.

Character sets and sets of state values are assumed to be enumerable if not stated otherwise. For any character set or alphabet A , $A^\epsilon := A \cup \{\epsilon\}$ where ϵ is the empty character. For state value sets Q , $Q^\epsilon := Q \cup \{\epsilon\}$ where ϵ is the undefined value. If either a character or state value set $A = A_1 \times \dots \times A_n$ is a Cartesian product then $A^\epsilon = A_1^\epsilon \times \dots \times A_n^\epsilon$.

Elements of character sets or state value sets can be vectors. There will be no notational distinction between single elements and vectors. However, a state vector (p_1, \dots, p_n) where p_k belongs to structure \mathcal{A}_k is written as \mathbf{p} and the change of this vector in a position k from p to q is written as $\mathbf{p} \left[\begin{smallmatrix} q \\ p \end{smallmatrix}, k \right]$. An n -dimensional vector of characters with the k -th component v and the rest ϵ is written as $\epsilon[v, k] = (\epsilon_1, \dots, \epsilon_{k-1}, v, \epsilon_{k+1}, \dots, \epsilon_n)$.

The power set of a set A is written as $\wp(A)$.

2 System definition

What allows us to draw these nice little boxes to represent systems versus the rest of the world? There seems to be a consensus (e.g. [22, 12, 55, 28, 27]) that a system separates an inside from the rest of the world, the environment. IEC 60050 [27] defines a system (351-42-08) as a "set of interrelated elements considered in a defined context as a whole and separated from their environment".

So, to gain a well defined system model we have to answer the two questions: What gets separated? And: what separates?

What gets separated? It's time dependent properties each taking a single out of a set of possible values at a given time. These time dependent functions are commonly called "state variables" and the values are called "states" [27]. However, in my opinion the term "variable" should better be restricted to the domain of descriptions, denoting the state functions. To make its meaning unique where necessary, I therefore will qualify the term "state" either with "function" or "value". If it is not qualified, I usually refer to its meaning as state function.

So, a state (function) s in this sense is a function from the time domain T to some set of state values or alphabet A or to a set of state values or alphabet A^ϵ , including the empty character or unknown value ϵ . Some of the states may not be directly accessible from the outside, these are the system's inner states. We usually assume that the inner states of a system are well defined, i.e. cannot attain the undefined value ϵ - but that states, representing the input or output of a system may attain the empty character ϵ .

What separates? Already in IEC 60050 [27] it is noted that a system is generally defined with the view of achieving a given objective, for example by performing a definite function. The key idea is now to use this function, that is the unique functional relation between the values of states at given time values to separate the state of a system from the rest of the world. Such a relation implies causality and a time scale.

Depending on the class of system function or time, different classes of systems can be identified. Important classes of functions are computable functions, finite functions and analytic functions. Important classes of times are discrete and continuous times¹.

A first example, intentionally not drawn from informatics, is a serial RLC-circuit. It consists of a resistor R , an inductor L , and a capacitor C and comes with a continuous time domain. Hence, it is described by a differential equation. I chose the current I and the voltage U_L across L as inner state, and the external voltage U_{ext} as input state. With the knowledge that the current is the same at every point in the circuit and that all voltages add up, the following two differential equations result for the time evolution of the two inner states:

$$\frac{dI(t)}{dt} = \frac{U_L(t)}{L} \quad \text{and} \quad \frac{dU_L(t)}{dt} = \frac{dU_{ext}(t)}{dt} - \frac{R}{L}U_L(t) - \frac{I(t)}{C}$$

In the following I will focus on discrete times. An example for a discrete system is a finite multiplier with its input state in_1, in_2 and its output state out , all ranging within $\{-2^{31} \dots 2^{31} - 1\}$ and the system function $out(t') = f(in_1(t), in_2(t)) = in_1(t) * in_2(t) \bmod 2^{31}$.

Hence, the formal system definition is as following.

Definition 1. *Be Q a non-empty set of internal state values and I and O the possibly empty sets of input and output state values (alphabets). The state*

¹ Referring to quantum physics we could also distinguish between classical and quantum states.

$(q, in, out) : T \rightarrow Q \times I \times O$ is said to form a (discrete) system for time step (t, t') , if it is aggregated by a function $f : Q \times I \rightarrow Q \times O$ with $f = (f^{int}, f^{ext})$ such that

$$\begin{pmatrix} q(t') \\ out(t') \end{pmatrix} = \begin{pmatrix} f^{int}(q(t), i(t)) \\ f^{ext}(q(t), i(t)) \end{pmatrix}.$$

t' is also called the successor time of t with respect to f .

The system time is the structure $\mathcal{T} = (T, succ)$ with T the enumerable set of all time values and $succ : T \rightarrow T$ with $succ(t) = t'$ is the successor function. For an infinite number of time values, T can be identified with the set of natural numbers \mathbb{N} . For a finite number of time values, $succ(t_{max})$ is undefined. I also write $t + n = succ^n(t)$.

A system thereby is characterized by a tuple $\mathcal{S} = (T, succ, Q, I, O, q, in, out, f)$. If Q, I and O are finite then the system is said to be finite. If T is enumerable, the system is said to be discrete. A system with no or only constant internal state is said to be stateless.

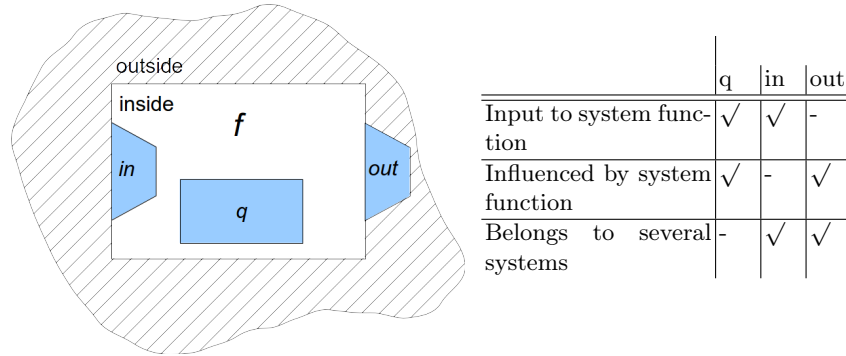


Fig.3. Diagram of a system as defined in Def. 1. The states (q, in, out) become separated from the rest of the world by the system function f . The asymmetric boxes of the input and output state symbolizes their asymmetric use in the system model. The input state in provides input to the system function but cannot be influenced by it. Both, input as well as output state can belong to different systems at a given point in time. The output state can be another system's input state and vice versa. In case, the inner state q or the system function are of no relevance for illustrative purposes, they can be omitted.

With definition 1 we can declare the meaning of these simple system diagrams as shown in Fig 3. As we will see, this kind of graphical representation is so intuitive, that it is perhaps one of the most stated "lies" in computer science. Too often, these kind of graphical boxes claim to represent "systems", but no system function with respect to the separated state functions can be identified nor probably any other formal system criteria can be fulfilled.

With the definition of functional equivalence, we create equivalence classes of isomorphic systems. To do so, we have to extend the system functions to operate also on O and map onto I .

Definition 2. Be S_1 and S_2 two systems and the extended system functions $f_k^* : Q_k \times I_k^\epsilon \times O_k^\epsilon \rightarrow Q_k \times I_k^\epsilon \times O_k^\epsilon$ with $f_k^*(q_k, i_k, o_k) = (f_k^{(int)}(q_k, i_k), 1_k(i_k), f_k^{(ext)}(q_k, i_k))$ with $k = 1, 2$.

S_1 and S_2 are isomorphic or functionally equivalent (written as $S_1 \cong S_2$) if two bijective functions $\phi : Q_1 \times I_1^\epsilon \times O_1^\epsilon \rightarrow Q_2 \times I_2^\epsilon \times O_2^\epsilon$ and $\psi : T_1 \rightarrow T_2$ exists such that $\phi(f_1^*(q_1, i_1, o_1)) = f_2^*(\phi(q_1, i_1, o_1))$ for each $(q_1, i_1, o_1) \in Q_1 \times I_1^\epsilon \times O_1^\epsilon$ and $\psi(\text{succ}_1(t)) = \text{succ}_2(\psi(t))$ for all $t \in T_1$.

Relating to these equivalence classes, we can restrict ourselves to systems operating on representations of natural numbers or a subset thereof.

2.1 Clocked and unclocked systems

An important differentiator for discrete systems is whether they are clocked or not. A clocked system applies its system function when the clock signal triggers it. An unclocked system applies its system function when an input arrives².

A clocked system thereby can be viewed as an unclocked system with an additional clock 'tick' as input. It can, in general, react if not to but on an empty input. This is the usual case for systems represented by computer programs running in the same address space.

As an unclocked system reacts only to an applied input signal, an empty input cannot trigger any processing. In this case, an empty input of a transition stands for a spontaneous action and an empty output represents no output at all. This is the usual case for systems interacting remotely without access to any common clock signal.

2.2 Computable functionality

Be F_n the set of all functions on natural numbers with arity n and we assume a set of initial computable functions (the successor, the constant and the identity function). Then, based on work of Kurt Gödel, Stephen Kleene [30] showed that there are three rules to create all computable functions:

1. *Comp*: Be $g_1, \dots, g_n \in F_m$ computable and $h \in F_n$ computable, then $f = h(g_1, \dots, g_n)$ is computable.

² Some authors like Rajeev Alur [1] name these system "synchronous" and "asynchronous". However, I would like to reserve these terms for the behavior of a "sending" system: a "synchronous" working system provides some output and waits for some input - the usual behavior of a component calling another component's function. An "asynchronous" working system provides some output and goes on processing without waiting for any further input coming back [47].

2. *PrimRec*: Are $g \in F_n$ and $h \in F_{n+2}$ both computable and $a \in \mathbb{N}^n$, $b \in \mathbb{N}$ then also the function $f \in F_{n+1}$ given by $f(a, 0) = g(a)$ and $f(a, b + 1) = h(a, b, f(a, b))$ is computable.
3. *μ -Rec*: Be $g \in F_{n+1}$ computable and $\forall a \exists b$ such that $g(a, b) = 0$ and the μ -Operation $\mu_b[g(a, b) = 0]$ is defined as the smallest b with $g(a, b) = 0$. Then $f(a) = \mu_b[g(a, b) = 0]$ is computable.

3 System composition

Systems can be composed from other systems. To compose, systems have to interact. In this theory systems interact by sharing common I/O-states: one system's output state is another system's input state. I call such a common state an "*ideal Shannon channel*", following the concept of Claude Shannon [54] that a channel reproduce at one point a state value selected at another point. As he showed, it is possible to reproduce any state value out of a finite set of state values over a channel even in the presence of noise up to any desired level of accuracy if sufficient time is available.

3.1 Compositionality

A precise system model allows for a precise definition of the notion of compositionality. Following an idea of Arend Rensink³, I distinguish between composition of systems and the property of being compositional for the properties of the systems.

Definition 3. *Be S the set of all systems according to Def. 1 and be $comp : S^n \rightarrow S$ a composition operator. A system property α is a partial function $S \rightarrow A$ which attributes values of some attribute set A to systems $S \in S$.*

A property α of a composed system $\mathcal{S} = comp(\mathcal{S}_1, \dots, \mathcal{S}_n)$ is called "compositional" if a function $comp_\alpha : A^n \rightarrow A$ exists that fulfills the following relation:

$$\alpha(comp(\mathcal{S}_1, \dots, \mathcal{S}_n)) = comp_\alpha(\alpha(\mathcal{S}_1), \dots, \alpha(\mathcal{S}_n)) \quad (1)$$

*Otherwise the system property is called "emergent". If $comp_\alpha$ is computable, then the property is called "computable compositional", otherwise it is called "computational emergent"*⁴.

In other words, compositional properties of the composed system result exclusively from the respective properties of the parts. Mathematically, α is a

³ Proposed in his talk "Compositionality huh?" at the Dagstuhl Workshop "Divide and Conquer: the Quest for Compositional Design and Analysis" in December 2012.

⁴ As something is "computable" means that there could be an algorithm to calculate it, being "computable emergent" would mean that the property of being emergent is computable - which it is not. Hence, I call it "computational" emergent, indicating, that from a computational perspective, it is emergent.

homomorphism. Emergent properties may result also from other properties α_i of the parts [43] as $\alpha(comp(\mathcal{S}_1, \dots, \mathcal{S}_n)) = comp_\alpha(\alpha_1(\mathcal{S}_1), \dots, \alpha_n(\mathcal{S}_n))$.

A simple example of a compositional system property is the mass $m(\mathcal{S})$ of a physical system \mathcal{S} . The total mass of the composed system is simply the sum of the masses of the single systems: $m(comp(\mathcal{S}_1, \dots, \mathcal{S}_n)) = m(\mathcal{S}_1) + \dots + m(\mathcal{S}_n)$.

An example for an emergent property in this sense is the resonance of the RLC-circuit. It's emergent and not compositional because the building blocks don't have a resonance frequency - only the RLC-circuit has.

3.2 Sequential system composition

First, I define what I mean by saying that two systems work sequentially and then I show that under these circumstances, a super system can always be identified. Sequential system composition means that one system's output is completely fed into another system's input, see Fig. 4.

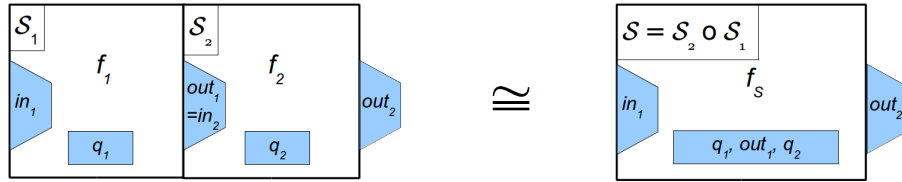


Fig. 4. Diagram of a sequential system composition as defined in Def. 1. The output state of the first system out_1 is identical to the input state in_2 of the second system.

Definition 4. Two discrete systems \mathcal{S}_1 and \mathcal{S}_2 with $O_1 \subseteq I_2$ and $\tau_i, \tau'_i \in T_i$ are said to "work sequentially" or to be "concatenated" at (τ_1, τ_2) if $out_1(\tau'_1) = in_2(\tau_2)$.

Proposition 1. Let \mathcal{S}_1 and \mathcal{S}_2 be two systems that work sequentially at time (τ_1, τ_2) . Then the states (q, in, out) with $q = (q_1, out_1, q_2)$, $in = in_1$, and $out = out_2$ form a system for time step $((\tau_1, \tau_2), (\tau'_1, \tau'_2))$.

Proof. To prove the proposition we have to identify a system function and a time function for the composed system. Based on the concatenation condition $out_1(\tau'_1) = in_2(\tau_2)$ we can replace $in_2(\tau_2)$ by $out_1(\tau'_1) = f_1^{ext}(q_1(t_1), i_1(t_1))$. We get

$$\begin{pmatrix} q_1(\tau'_1), out_1(\tau'_1), q_2(\tau'_2) \\ out_2(\tau'_2) \end{pmatrix} = \begin{pmatrix} f_1(q_1(\tau_1), i_1(\tau_1)), f_2^{int}(q_2(\tau_2), f_1^{ext}(q_1(\tau_1), i_1(\tau_1))), out_2(\tau_2)) \\ f_2^{ext}(q_2(\tau_2), f_1^{ext}(q_1(\tau_1), i_1(\tau_1))), out_2(\tau_2) \end{pmatrix} \quad (2)$$

where the right hand side depends exclusively on τ_1 and τ_2 and the left hand side, the result, exclusively on τ'_1 and τ'_2 .

The composed system C exists with the time values $T_C = \{0, 1\}$. With the functions $map_i : T_C \rightarrow T_i$, ($i = 1, 2$) with $map_i(0) = \tau_i$ and $map_i(1) = \tau'_i$, the states of C are: $out_C(\tau_C) = out_2(map_2(\tau_C))$, $in_C(\tau_C) = in_1(map_1(\tau_C))$, and $q_C(\tau_C) = (q_1(map_1(\tau_C)), out_1(map_1(\tau_C)), q_2(map_2(\tau_C)))$.

The concatenation condition $out_1(\tau'_1) = in_2(\tau_2)$ implies that there is an important difference between the inner and outer time structure of sequentially composed systems. To compose sequentially, the second system can take its step only after the first one has completed its own.

Thus, the concatenation operator $\circ_{(\tau_1, \tau_2, out_1, in_2)}$ ⁵ composes systems such that $\mathcal{S} = \mathcal{S}_2 \circ_{(\tau_1, \tau_2, out_1, in_2)} \mathcal{S}_1$ is the concatenated or sequentially composed system for one time step. The extension to more than two concatenated systems or time steps is straight forward.

In the special case, where $f_i^{ext} : I_i \rightarrow O_i$ is stateless in both systems, the concatenation of these stateless systems results in the simple concatenation of the two system functions: $out_2(\tau'_2) = (f_2^{ext} \circ f_1^{ext})(in_1(\tau_1))$

Finally, the following proposition holds:

Proposition 2. *For sequential system composition, the system's functionality composes compositionally.*

Proof. Identifying α of the compositionality definition 3 with the mapping from the domain of the systems to the domain of the system functions, we see that $f_{\mathcal{S}_2 \circ \mathcal{S}_1} = \alpha(\mathcal{S}_2 \circ \mathcal{S}_1) = comp_\alpha(\alpha(\mathcal{S}_1), \alpha(\mathcal{S}_2)) = comp_\alpha(f_{\mathcal{S}_1}, f_{\mathcal{S}_2})$. The wanted operator $comp_\alpha$ can be directly deduced from the system function as defined in Eq. 2.

Fig. 5 shows an example of an additional interaction, influencing system composition behavior and illustrates the suggestibility of visual descriptions.

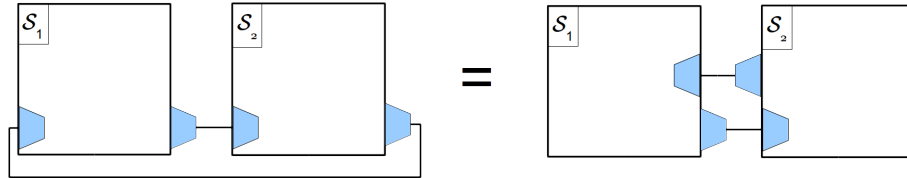


Fig. 5. This is an example of visual suggestibility. The left hand side suggest a sequential system composition with an additional feedback channel. The right hand side suggests that we have two reciprocally coupled systems. Actually, both diagrams convey exactly the same information.

⁵ If the time steps and the concatenation states are clear from the context, then they are omitted in the notation.

Especially important, neither diagram of Fig. 5 conveys any information about the timing of the coupling. If the diagrams are complete and there is no additional input to the systems, then, the resulting "thing" is no system any longer, as it has neither a dedicated input nor a dedicated output state. Actually, if both systems are unclocked, nothing will happen anyhow.

How can it be that linking the output of one system to the input of another system results in sequential composition but creating another link destroys that? Actually, it does not "destroy" the sequential composition. The additional link just shows the importance of the timing, that is the assignment of time steps to the evaluation of the system function. Assuming clocked systems, we can still see that \mathcal{S}_1 is doing something and hands over its character to \mathcal{S}_2 and say that the interaction resulted in a sequential system composition for these two time steps. But in the next step, its again \mathcal{S}_1 doing something. So for these three consecutive actions, we do not have a sequential composition anymore. What do we have instead? There is no simple answer to this seemingly simple question, as I will discuss in section 3.5 and 4.

3.3 Parallel system composition

Parallel processing systems can also be viewed as one system if there is a common input to them and the parallel processing operations are independent and therefore well defined as is illustrated in Fig. 6.

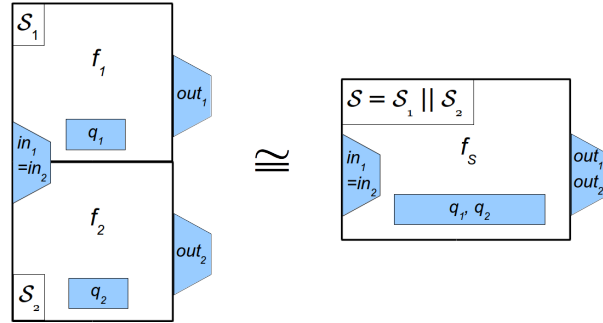


Fig. 6. Diagram of a parallel system composition as defined in Def. 5. The input of both systems is identical.

Definition 5. Two discrete systems \mathcal{S}_1 and \mathcal{S}_2 with $I_1 = I_2$ are said to "work in parallel" at (τ_1, τ_2) if $q_1 \neq q_2$, $out_1 \neq out_2$ and $\tau_1 \in T_1$, $\tau_2 \in T_2$ exist such that $in_1(\tau_1) = in_2(\tau_2)$.

Proposition 3. Let \mathcal{S}_1 and \mathcal{S}_2 be two systems that work in parallel at (τ_1, τ_2) . Then the state (q, in, out) , with $q = (q_1, q_2)$ $in = in_1 = in_2$, $out = (out_1, out_2)$ form a system for time step $((\tau_1, \tau_2), (\tau'_1, \tau'_2))$.

Proof. To prove, we have to provide a system function for the composed system C at times $T_C = \{0, 1\}$, which is simply $f_C : (Q_1 \times Q_2) \times (I_1^\epsilon \times I_2^\epsilon) \rightarrow (Q_1^\epsilon \times Q_2^\epsilon) \times (O_1^\epsilon \times O_2^\epsilon)$ with

$$f_C(\tau'_C) = \left(f_1^{int}(q_1(\text{map}_1(\tau_C)), \text{in}(\text{map}_1(\tau_C))), f_1^{ext}(q_2(\text{map}_2(\tau_C)), \text{in}(\text{map}_1(\tau_C))) \right) \\ \left(f_2^{int}(q_1(\text{map}_1(\tau_C)), \text{in}(\text{map}_1(\tau_C))), f_2^{ext}(q_2(\text{map}_2(\tau_C)), \text{in}(\text{map}_1(\tau_C))) \right)$$

The composed system's time step does not depend on the exact sequence of the time steps of the subsystems. If both systems don't perform their step at the same time compared to an external clock then the composed system does not realize its time step in a synchronized manner. If this synchronization takes place, then there is effectively no difference between the inner and outer time structure of parallel composed systems.

The parallelization operator $\|_{(\tau_1, \tau_2, in_1, in_2)}$ can be defined such that $\mathcal{S} = \mathcal{S}_2 \|_{(\tau_1, \tau_2, in_1, in_2)} \mathcal{S}_1$ is the parallel working system for one time step. The extension to more than two parallel working systems or two time steps is again straight forward.

As with sequential system composition,

Proposition 4. *for parallel system composition, the system's functionality composes compositionally.*

The proof is similar to that for sequential system composition.

3.4 Combining sequential and parallel system composition

As the application of an operation f preceding two parallel operations g and h is equivalent to the parallel execution of g after f and h after f , together with some bookkeeping on time steps and i/o-states, the following proposition is easy to prove:

Proposition 5. *For three systems \mathcal{P}_1 , \mathcal{P}_2 and \mathcal{S} where \mathcal{P}_1 and \mathcal{P}_2 are parallel systems and $\mathcal{P}_1 \| \mathcal{P}_2$ and \mathcal{S} are sequential systems, then the two systems $(\mathcal{P}_1 \| \mathcal{P}_2) \circ \mathcal{S}$ and $(\mathcal{P}_1 \circ \mathcal{S}) \| (\mathcal{P}_2 \circ \mathcal{S})$ are functional equivalent, that is $(\mathcal{P}_1 \| \mathcal{P}_2) \circ \mathcal{S} \cong (\mathcal{P}_1 \circ \mathcal{S}) \| (\mathcal{P}_2 \circ \mathcal{S})$. In other words, the right distribution law holds for parallel and sequential composition with respect to functional equivalence.*

As sequential system composition is non-commutative, it follows that the left distribution law does not hold: $\mathcal{S} \circ (\mathcal{P}_1 \| \mathcal{P}_2) \not\cong (\mathcal{S} \circ \mathcal{P}_1) \| (\mathcal{S} \circ \mathcal{P}_2)$.

Finally I point out that combining parallel and sequential system composition as $\mathcal{S} \circ (\mathcal{P}_1 \| \mathcal{P}_2)$ for stateless systems results in a combination of their system functions as $f_{\mathcal{S}}(f_{\mathcal{P}_1}, f_{\mathcal{P}_2})$, which is exactly the way, the first rule *Comp* for computable functions was defined. Hence, from a formal point of view, for computational systems, sequential together with parallel system composition represent the first level to compose computable functionality.

3.5 Recursive system composition

From an interaction perspective we could say that recursive interaction means that at least two systems send each other characters back and forth. But this kind of character ping-pong does not say very much about the created logical system relations as is illustrated in Fig 7.

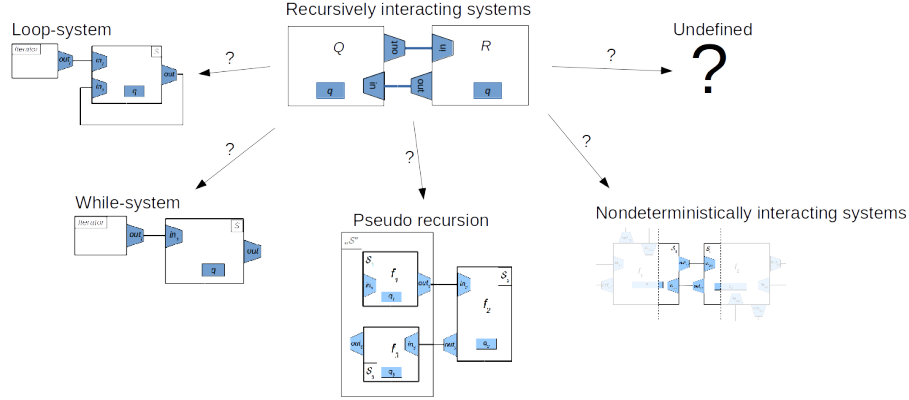


Fig. 7. In the middle, two systems interact by sending each other characters. What does that mean from a system perspective? Does a loop- or a while-system result? Or is the interaction loosely coupled? Or is it only pseudo recursive? Or does something else happens?

As sequential together with parallel system composition represent the first level to compose computable functionality, it is interesting to investigate how the system composition operators have to look like for the other two rules *PrimRec* and μ -*Rec*.

For *PrimRec* we have to sequentially combine an iterator together with a system which has its output as its own input, see Fig. 8. Additionally, some mechanism has to terminate the computation after the n -th step.

Definition 6. Be g and h two functions as defined in *PrimRec*:. Be \mathcal{I} a discrete clocked system with with $I = \emptyset$, $Q = \{0, \dots, n+1\}$, $O = \{0, \dots, n\}$, $f : Q \rightarrow Q \times O$ with $(q', out') = f(q) = (q+1, q)$, for $t < n$. And \mathcal{S} is a discrete (not necessarily clocked) system with $I = \{0, \dots, n\} \times \mathbb{N}$, $Q = \{a\}$, $O = \mathbb{N}$, $f : Q \times I \rightarrow O$ with $f(q, in) = h(q, in)$ are said to work in a for-loop for n times if $(\tau_{\mathcal{I}}, \tau_{\mathcal{S}})$ exist, such that $out_{\mathcal{I}}(\tau_{\mathcal{I}} + i + 1) = in_{1\mathcal{S}}(\tau_{\mathcal{S}} + i)$, $out_{\mathcal{S}}(\tau_{\mathcal{S}} + i) = in_{2\mathcal{S}}(\tau_{\mathcal{S}} + i)$.

Proposition 6. Let \mathcal{I} and \mathcal{S} be two systems as defined in Def. 6 and $(\tau_{\mathcal{I}}, \tau_{\mathcal{S}}) \in T_{\mathcal{I}} \times T_{\mathcal{S}}$ such that $out_{\mathcal{I}}(\tau_{\mathcal{I}}) = 0$, $q_{\mathcal{S}}(\tau_{\mathcal{S}}) = a$, $in_{\mathcal{S},2}(\tau_{\mathcal{S}}) = out_{\mathcal{S}}(\tau_{\mathcal{S}}) = g(a)$, and $in_{\mathcal{L},2} = n$. Then the state $in_{\mathcal{L}} = q_{\mathcal{S}}$, $out_{\mathcal{L}} = out_{\mathcal{S}}$ form a system for time step $((\tau_{\mathcal{I}}, \tau_{\mathcal{S}}), (\tau_{\mathcal{I}} + n, \tau_{\mathcal{S}} + n))$. We also write $\mathcal{L} = Loop_{a,n}(\mathcal{I}, \mathcal{S})$

Proof. That the loop system calculates $f(a, n)$ of *PrimRec* in the $n - th$ time step can be deduced from the following table.

$time_{\mathcal{I}}$	$out_{\mathcal{I}}(\tau_{\mathcal{I}} + i)$	$time_{\mathcal{S}}$	$in_{\mathcal{S},2}(\tau_{\mathcal{S}} + i)$	$in_{\mathcal{S},2}(\tau_{\mathcal{S}} + i)$ $= out_{\mathcal{S}}(\tau_{\mathcal{S}} + i)$
$\tau_{\mathcal{I}} + 0$	0	-	-	-
$\tau_{\mathcal{I}} + 1$	1	$\tau_{\mathcal{S}} + 0$	0	$f(a,0) = g(a)$
$\tau_{\mathcal{I}} + 2$	2	$\tau_{\mathcal{S}} + 1$	1	$f(a,1) = h(a, 0, g(a))$
$\tau_{\mathcal{I}} + 3$	3	$\tau_{\mathcal{S}} + 2$	2	$f(a,2) = h(a, 1, h(a,0,g(a)))$
\dots	\dots	\dots	\dots	\dots

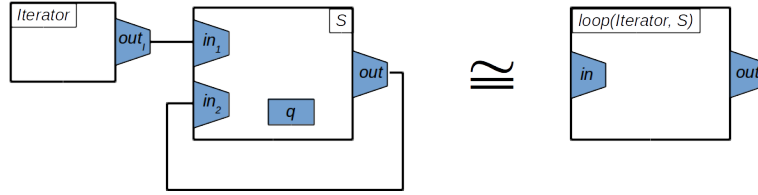


Fig. 8. Diagram of a loop system composition as defined in Def. 6.

Please note that there is no explicit end of the loop counter, but the loop has to be terminated by some external mechanism - or to put it the other way around. If some external mechanism guarantees termination of the loop after n steps, then we can always identify a system with a well defined time step.

Also, the following proposition holds:

Proposition 7. *For a loop-system according to Def. 6, the system function composes compositionally.*

Proof. Be $\alpha(\mathcal{S})$ the system function, then compositionality means that the system function of the loop system of \mathcal{I} and \mathcal{S} can be constructed from the system functions of \mathcal{I} and \mathcal{S} together with the additional knowledge of the system construction which comprise n and the state identity $out_{\mathcal{I}} = in_{1\mathcal{S}}$, $out_{\mathcal{S}} = in_{2\mathcal{S}}$ which can be incorporated into $comp_{\alpha}$. This is the case as can be seen from the proof of proposition 6.

For $\mu\text{-Rec}$, we again combine an iterator together with another system, whose only task is to calculate the zeros of g as is illustrated in Fig. 9. Again, some mechanism has to explicitly stop the computation after the zero has been computed.

Definition 7. *Be g a function as defined in $\mu\text{-Rec}$. Two discrete Systems \mathcal{I} with $I_{\mathcal{I}} = \emptyset$, $Q_{\mathcal{I}} = O_{\mathcal{I}} = \mathbb{N}$, $f : Q_{\mathcal{I}} \rightarrow Q_{\mathcal{I}} \times O_{\mathcal{I}}$ with $(q'_{\mathcal{I}}, out'_{\mathcal{I}}) = f_{\mathcal{I}}(q_{\mathcal{I}}) = (q_{\mathcal{I}} + 1, q_{\mathcal{I}})$ and \mathcal{S} with $I_{\mathcal{S}} = \mathbb{N}$, $Q_{\mathcal{S}} = \{a\}$, $O_{\mathcal{S}} = \mathbb{N}$, $f_{\mathcal{S}} : I_{\mathcal{S}} \times Q_{\mathcal{S}} \rightarrow O_{\mathcal{S}}$ with*

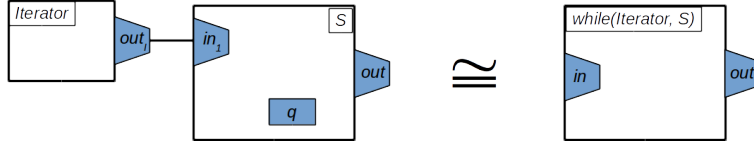


Fig. 9. Diagram of a while system composition as defined in Def. 7.

$f_S(in_S, q_S) = g(q_S, in_S)$ are said to work in a while-loop if (τ_I, τ_S) exist, such that $out_I(\tau_I + i + 1) = in_S(\tau_S + i)$.

Proposition 8. Let \mathcal{I} and \mathcal{S} be two systems as defined in Def. 7 and $(\tau_I, \tau_S) \in T_I \times T_S$ such that $out_I(\tau_I) = 0$ and $q(\tau_S) = a$. Then the states $in_{\mathcal{L}} = q_S$ and $out_{\mathcal{L}} = out_I$ form a system if there is a $\delta \in \mathbb{N}$ such that $f_S(\delta, q) = 0$. Then the time step is $((\tau_I, \tau_S), (\tau_I + \delta, \tau_S + \delta))$. We also write $\mathcal{L} = While_a(I, S)$

The proof is similar to the previous ones. For while-systems the following proposition holds:

Proposition 9. The system function of a while-composition composes computationally emergently.

Proof. Even if the while functionality composes compositionally, for a system, according to definition 1 we have to provide a time step. However, the knowledge, that g somehow has a zero does not suffice. We would need to have a general algorithm to calculate the specific δ , which does not exist, hindering us to provide the time step.

As both, the loop- as well as the while system are stateless, another immediate consequence of Kleene's recursive function theorem is the next proposition:

Proposition 10. Every computable stateless system is functionally equivalent to a system of the form $\mathcal{S} \circ (\mathcal{P}_1 || \dots || \mathcal{P}_n)$, $Loop_n(I, S)$ or $While(I, S)$.

This shows that from a functional perspective, "state" is only needed to represent input and output values and it is also needed internally for loop- and while-computations, but "state" is not needed for the representation of computational functionality. This fact forms the basis for the influential area of "functional programming" [26] where manipulating state is termed a "side-effect". A point of view which perpetuates itself until nowadays in the form of the so called "REST-architecture paradigm" [17].

But as we will see below, "state" does become very important for cooperative interactions and hence, any approach to neglect it will have its weaknesses in describing these kind of system relations.

Equipped with a notion of system composition, the concept of super- or subsystem can be defined.

Definition 8. Let \mathcal{S} be composed from systems \mathcal{U}_k ($k = 1, \dots, n$) according to definition 1, 5, 6, or 7. Then \mathcal{S} is called the super system of the \mathcal{U}_k and the \mathcal{U}_k are the subsystems of \mathcal{S} .

3.6 Pseudo recursion or U-composition

A very common "pattern" of system composition is shown in Fig 10. Of three consecutive systems, the first and last are attributed to a "system \mathcal{S} " which seems to interact recursively with, that is providing output to and receiving input from, the intermediate system \mathcal{S}_2 .

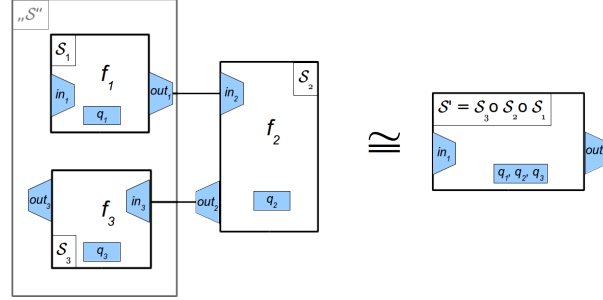


Fig. 10. Diagram of a system sequence. Very often there is a box drawn around system \mathcal{S}_1 and \mathcal{S}_3 and this box is treated as a system " \mathcal{S} " as is shown in gray. But this is - although very intuitive - an example of graphical lie. Indeed, all three systems \mathcal{S}_1 , \mathcal{S}_2 , and \mathcal{S}_3 are subsystems of the resulting system $\mathcal{S}' = \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1$.

Actually, \mathcal{S} could be viewed as $\mathcal{S}_1 || \mathcal{S}_3$ - if both systems shared their input and execute in parallel - which obviously is not the case. So, what is termed as " \mathcal{S} " is not a system in the sense of Def. 1. In fact, all three systems \mathcal{S}_1 , \mathcal{S}_2 , and \mathcal{S}_3 are subsystems of the resulting system $\mathcal{S}' = \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1$.

But we will see that this pattern is of great importance, when it comes to functional components, as both, " \mathcal{S} " and \mathcal{S}_2 , can be viewed as components of \mathcal{S}' . I also speak of U-composition.

4 System cooperation

As was illustrated in Fig. 7, from a composition perspective, it is not a priori clear what really happens if systems mutually interact. One important class of interactions that was already mentioned in Fig. 7 but does not lead to system composition in the sense of section 3 are nondeterministic, stateful, asynchronous interactions⁶.

The focus is shifted from hierarchical system composition dominated by an inductive notion of a system function to two complementary issues: First, how can we describe the nondeterministic interactions? How much of the systems do we have to include, how much can we exclude? And second, how can we describe

⁶ An interaction classification, based on these three parameters can be found in [47]

a deterministically working system, where an action is an execution of its system function, but which takes part in multiple nondeterministic interactions? We will see that this leads to the question, how to deal with the problem of interaction coordination.

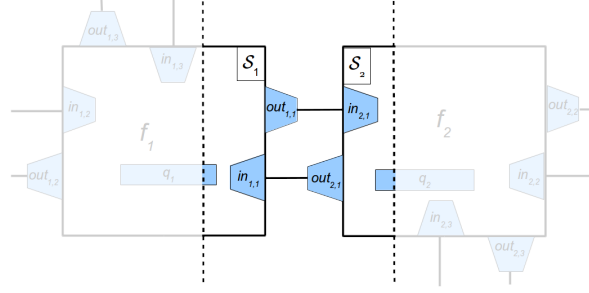


Fig. 11. Two systems, S_1 and S_2 interacting with each other and with multiple other systems. Within a single interaction, only the projection of the system functionality onto the interaction channels becomes relevant.

In contrast to a function which represents a unique mapping, a nondeterministic relation between input and output values prompts for an additional criterion to distinguish between success and failure. We also do not expect a successful cooperation to be attributed to a single system, but to all involved systems. Also, we need a tool with which we can describe the relevant parts of the involved systems while ignoring the irrelevant ones.

The first issue, how we should describe the nondeterministic interactions, is solved by (partitioned) I/O-automata coupled by ideal Shannon Channels. I/O-automata fulfill both requirements, to provide an additional acceptance component and to be able to represent only the, for the interaction, relevant parts of a system. Their coupling by Shannon channels restricts their transition relation and leads to the protocol concept (e.g. [25, 9, 34]) as the outer coupling of the interacting systems. Here systems in given roles mutually document their state transitions in a way that is comprehensible to the respective receivers and thereby all systems can achieve a common goal together. As is illustrated in Fig. 11, only projections of the interacting systems onto the interaction channels become visible. The knowledge of the interaction does not suffice anymore to determine the actions of the interaction partners from the perspective of the sender.

The second issue of interaction coordination is solved by using the building blocks of the interactions, namely the roles, and by searching for a complementary inner coupling which represents the desired causal linkage between the transitions of the different roles of a given system. That is, we look for logical

conditions to restrict the transition relation of the product of all roles until determinism is established and thereby a well defined system function is provided.

An extensive description illustrated with the example of a resource administering process can be found in [46]. Here I present only the most important, slightly modified definitions.

4.1 The building blocks: input/output automata

Input/output automata will be used to describe both, processes and their interactions.

Definition 9. A "nondeterministic I/O automaton (NIOA)" is a tuple $\mathcal{A} = (Q, I, O, q_0, Acc, \Delta)$. Q is the non-empty set of state values, I and O are the, possibly empty, input and output alphabets. q_0 is the initial state value, Acc is the acceptance component and $\Delta \subseteq Q \times Q \times I^\epsilon \times O^\epsilon$ is the transition relation. Instead of $(p, q, i, o) \in \Delta$ we also write $p \xrightarrow{i/o} q$. In case Q , I , and O are finite, then the automaton is called "finite" (NFIOA).

The acceptance component Acc represents the information needed to express "successful" computation, that is, the acceptance condition. It depends on the computational model of "success". For a finite computation Acc_{finite} is a finite set of final state values. For an infinite computation of a finite automaton, a representative acceptance condition is to accept all runs in which the finite set of infinitely often occurring state values is an element of the acceptance component (Muller-acceptance), that is for $Acc_{Muller} \subseteq \wp(Q)$ [16].

Transitions with an ϵ -character as input are called *spontaneous*. In case that for each $(p, i) \in Q \times I^\epsilon$ there is at most one transition $(p, q, i, o) \in \Delta$ then Δ specifies a function $\delta : Q \times I^\epsilon \rightarrow Q \times O^\epsilon$ with $(q, o) = \delta(p, i)$. If no ϵ is used, that is $\delta : Q \times I \rightarrow Q \times O$, we have a deterministic I/O-automaton or *DIOA*. A finite DIOA is a Mealy automaton [37].

An automaton is supposed to be executed in a certain way. It is important that some properties of an NIOA depends on the model of execution. It could well be that an automaton permanently provides a character as a possible input, but the execution model never chooses it - which is called "starvation".

Definition 10. Be \mathcal{A} an NIOA. A sequence q_0, q_1, \dots, q_n where $q_0 = q_{0,\mathcal{A}}$ is the initial state and $(q_{j-1}, q_j, i, o) \in \Delta_{\mathcal{A}}$ is called an "execution". An infinite sequence is called an " ω -execution". The pair (q_{j-1}, q_j) is called an event.

A computation of an execution is performed according to the following rules:

1. The first state value is the initial state of \mathcal{A} , q_0 .
2. Evaluate the current state value with respect to the acceptance component
3. With p as the current state value, arbitrarily choose one of the possible transitions $(p, q, i, o) \in \Delta_{\mathcal{A}}$ and thereby determine the input characters $i \in I_{\mathcal{A}}^\epsilon$.
4. q is the new current state value of the execution. Resume with step 2

An ω -execution is called *weakly fair* if a constantly available input character is eventually chosen. An ω -execution is called *strongly fair* if a repeatedly possible input character is eventually chosen.

The reason to introduce NIOAs is their ability to represent projections of systems.

Definition 11. An NIOA \mathcal{A} "specifies a projection" of a system \mathcal{S} , if $Q_{\mathcal{S}} \subseteq Q_{\mathcal{A}}$, $I_{\mathcal{S}} \subseteq I_{\mathcal{A}}$, $O_{\mathcal{S}} \subseteq O_{\mathcal{A}}$, and a projection function⁷ $\pi = (\pi_Q, \pi_I, \pi_O) : Q_{\mathcal{S}} \times I_{\mathcal{S}}^\epsilon \times O_{\mathcal{S}}^\epsilon \rightarrow Q_{\mathcal{A}} \times I_{\mathcal{A}}^\epsilon \times O_{\mathcal{A}}^\epsilon$ exists such that $\Delta_{\mathcal{A}}$ is the smallest possible set and for each time pair $(t, t') \in T \times T$ in every possible sequence, $(\pi_Q(q(t)), \pi_Q(q(t')), \pi_I(in(t)), \pi_O(out(t')))) \in \Delta_{\mathcal{A}}$.

It is clear that to any system according to Def. 1 there is a corresponding deterministic I/O-automaton (DIOA) and to any DIOA there is a corresponding system. If the system has only one input and output state function, then the projection onto this states reproduces the complete system.

The precondition of any coupling of NIOAs is to view them together as a product automaton, stepping synchronously and where all acceptance conditions are to be fulfilled simultaneously.

Definition 12. The weakly synchronized product of a set of n NIOAs \mathcal{A}_k is defined by NIOA $\mathcal{B} = (Q, I, O, \mathbf{q}_0, Acc, \Delta)_{\mathcal{B}}$, with $Q_{\mathcal{B}} = \times Q_k$, $I_{\mathcal{B}} = \times I_k$, $O_{\mathcal{B}} = \times O_k$, $\mathbf{q}_{0\mathcal{B}} = (q_{01}, \dots, q_{0n})$, the common acceptance component represents the logical conjunction of the individual components, symbolized as $Acc_{\mathcal{B}} = \bigwedge Acc_k$, $\Delta_{\mathcal{B}} := \{(\mathbf{p}, \mathbf{q}, \mathbf{i}, \mathbf{o}) \mid \text{the components of } \mathbf{p} \text{ are reachable states of the } \mathcal{A}_i \text{ and } \mathcal{A}_k \text{ provides a transition } (p_k, q_k, i_k, o_k) \text{ with } \mathbf{q} = \mathbf{p} \left[\begin{smallmatrix} q_k \\ p_k \end{smallmatrix}, k \right] \text{ and } \mathbf{i} = \epsilon[i_k, k] \text{ and } \mathbf{o} = \epsilon[o_k, k] \}$. I also write $\mathcal{B} = \bigotimes_{i=1}^n \mathcal{A}_i$.

We can say that for a state transition $(p, q, i, o) \in \Delta$ the event (p, q) is elicited by i and documented by o . This kind of "event documentation" is an inherent part of the transition semantic.

Please note, that the definition 12 of a weakly synchronized product automaton, which will be the basis for the protocol definition, entails an important restriction with respect to its I/O-behavior compared to a system: it restricts its input or output to a single component of its I/O-vectors (which by themselves could be vectors). The definition of a system does not provide any restrictions on actuating its output components of being driven by multiple input components - the weakly synchronized product automaton does so.

Partitioned I/O-automata With the help of an equivalence relation \equiv on transitions, the transition relation Δ can be partitioned into (disjoint) equivalence classes Δ_l with $l \in L \subseteq \mathbb{N}$. Then there is a function $part : \Delta \rightarrow L$ which maps each tuple of the transition relation to the index of its equivalence class.

There are two important equivalence class constructions which are both based on descriptive equivalence, that is two transitions are equivalent according to a given description.

⁷ A projection function π is defined by the property $\pi = \pi \circ \pi$.

Extended Automata For nondeterministic I/O-automata, we construct the equivalence relation \equiv_{ND} as following:

1. Be $DocCls$ a set of document classes and $Param$ a set of parameter values. Then be $parse : I \cup O \rightarrow DocCls \times Param$ with $(docCls, param) = parse(a)$ a function assigning a document class and parameter values to each character. We can thereby look at each character as an instance of a document of a given document class representing parameter values.
2. All state values have a mode component and a rest: $Q = Q_{mode} \times Q_{rest}$.
3. Be $Cond$ a set of conditions: $Cond : Q_{rest} \times Param \rightarrow \{true, false\}$.

Then, two transitions $t, t' \in \Delta$ are equivalent $t \equiv_{ND} t'$ if both can be attributed the same $docCls_i, docCls_o, p_{mode}, q_{mode}$, and $cond(p_{rest}, param_i)$, that is, if their value of the partition function, set up as

$$part(p, q, i, o) = part(docCls_i, docCls_o, p_{mode}, q_{mode}, cond(p_{rest}, param_i))$$

is identical. p_{rest} as well as $param_o$ remain unconsidered. Instead of Δ_l we also write

$$p_{mode} \xrightarrow{docCls_i, cond(p_{rest}, param_i) / docCls_o} q_{mode} \quad (3)$$

An example is a seller that gets an *Order* from a customer, transits from *listening* to *ordered* if the customer is evaluated as being *trustworthy* and sends back a *Confirmation*:

$$listening \xrightarrow{Order, isTrustworthy(Customer) / Confirmation} ordered$$

So, "extended" automata, as they are often called, are actually not so much extended but rather "abstracted" by equivalence class construction. Please note that in contrast to the notation $p \xrightarrow{i/o} q$ which denotes a single transition, the notation of the expression 3 refers to an equivalence class of transitions.

Objects/Abstract Data Types In the case of deterministic I/O-automata, also the subautomata are deterministic and each subrelation Δ_l specifies a function $\delta_l : I \times Q \rightarrow O \times Q$ with $(o, q) = \delta_l(i, p)$ for $(p, q, i, o) \in \Delta_l$. If we treat all I/O-characters as state values then all state transitions relate to externally provided "data"-inputs.

We have $(o, q) = \delta_l(i, p) = \delta_l|_p(i)$. Indeed these are two equations:

$$o = \delta_l^{(o)}|_p(i) \quad (4)$$

$$q = \delta_l^{(q)}|_p(i) \quad (5)$$

The first equation represents the object oriented way of describing state transitions by providing a name for the state function tuple, the object name: `outputParameters = objectName.method(inputParameters)`. The second equation is only implicitly given in the world of objects.

If we additionally assume all state values to be partitioned into a mode component and a rest: $Q = Q_{mode} \times Q_{rest}$, then we have three equations:

$$o = \delta_l^{(o)}|_{p_{mode}}(i, p_{rest}) \quad (6)$$

$$q_{mode} = \delta_l^{(mode)}|_{p_{mode}}(i, p_{rest}) \quad (7)$$

$$q_{rest} = \delta_l^{(rest)}|_{p_{mode}}(i, p_{rest}) \quad (8)$$

$$(9)$$

The definition of a mode-state is known as "state pattern" in the object oriented world [18] but is usually not directly syntactically supported. It allows the definition of generic events (p_{mode}, q_{mode}) . But, in contrast to the state transition $p \xrightarrow{i/o} p$ mentioned before, the documentation of these events is not part of the transition semantic!

There is a special case of nondeterminism which is treated similar to determinism: exceptions. With exceptions we can syntactically distinguish between a desired "normal" functioning and undesired "exceptional" circumstances. An example is the command to write to a file system. Usually it will succeed - except, when the disk is full. So, for most purposes we can assume a deterministic relation and for the rare circumstances, our assumption proves to be wrong, we have to fundamentally change our proceeding anyway. To create exceptions, the transition relation has to be partitioned into two main parts: a deterministic part which gets further partitioned into a set of subrelations as described for the DIOA. And a rest, which gets partitioned into additional - usually only few - equivalence relations, representing the exceptional circumstances.

A possible way of syntactically expressing these equivalence classes is the well known exception mechanism, where the execution model is extended by a try-catch-mechanism.

4.2 Outer coupling: channel based restriction

The outer coupling is concerned with the causal connection between the output and input of different systems, which I call "outer coupling", and which was illustrated in Fig. 11. The essential question is: How can we describe the non-deterministic interaction between deterministic systems. We will aim for a notion, which we call "protocol", which is an NIOA that does not receive any additional input - as it already represents all the input and output of the involved systems.

With respect to systems, a Shannon channel is a common I/O-state function as defined in section 3. In the context of I/O-automata representing system projections, a channel does two things: first, it creates a product automaton, and second, it restricts the set of reachable state values of the product automaton by modifying the rules for executing the automaton's transition rules⁸. If a character

⁸ In [46] the same effect was accomplished by introducing "excited" state values. However, I think that relating to a modified execution is a more adequate way of comprehension.

is output on a Shannon channel it has to be processed as an input character in the next step.

Definition 13. Be \mathcal{A} an NIOA where O_k is the k -th component of the output alphabet $O = O_1 \times \dots \times O_n$ and I_l is the l -th component of the input alphabet $I = I_1 \times \dots \times I_m$ with $O_k \subseteq I_l$. The pair (k, l) is called a Shannon channel if it modifies the execution rules for \mathcal{A} in the following way:

1. Initially, the automaton is in its initial state q_0 .
2. Evaluate the current state value with respect to the acceptance component
3. Be $p \in Q_{\mathcal{A}}$ a reachable state of \mathcal{A} which was reached by a transition with the k -th output component $o \in O_k$. Then only a transition of $\Delta_{\mathcal{A}}$ can be chosen next which starts from p and has $o \in I_l$ as input character. I call the transition providing the character sending transition, the ensuing transition receiving transition.
4. Be $p \in Q_{\mathcal{A}}$ either the initial state or a reachable state of \mathcal{A} which was reached by a transition with a character $o \in O_{\mathcal{A}}$ with $o_k = \epsilon$ that is no output on the channel. Then only a transition of $\Delta_{\mathcal{A}}$ can be chosen next starting from p with ϵ as its l -th component, that is, which is either spontaneous or which has an input not associated with the channel.
5. q is the new current state value of the execution. Resume with step 2

Such an execution is also called a "channel based restricted (CBR-)" execution. The automaton with all reachable states of \mathcal{A} and only the executable transitions of \mathcal{A} under CBR is called a "CBR-automaton".

A CBR-automaton where all input and output states are connected via Shannon channels is called "closed", otherwise it is called "open".

It is not a priori clear that this procedure creates something meaningful. A condition which simplifies the execution of a CBR-automaton is that at most one character has to be considered as input. Hence, the ability of a system to output a vector of characters where multiple positions of this vector are different from ϵ is to be avoided. This is an important difference to the general definition of a system and its corresponding DIOA.

Definition 14. A CBR-automaton is called "linear-executable" if for each element of I and O at most one component is unequal to ϵ . Otherwise it is called "tree-executable"

For tree-executable CBR-automata, the result of the execution will in general depend on the execution strategy, for example which branch of the execution tree is calculated first.

Another necessary condition obviously is that there has to be a receiving transition for each sending transition for each channel, that each interaction chain terminates and that the acceptance condition can still be met. I therefore define:

Definition 15. A CBR automaton is called "well formed" if for every channel mediated transition which sends a character (different from ϵ) there exists a receiving transition to process it.

This is a typical safety property (something bad never happens). To automatically check it, all reachable states have to be explored. Hence, for infinite automata it is generally undecidable. For finite automata it is decidable, but its solution requires exponential effort with the number of state functions.

Definition 16. A well formed channel based restricted automaton is called "consistent" if for each reachable state value either the acceptance condition is met or there is at least one continuation such that the acceptance condition can be met and every interaction chain eventually terminates.

This is a typical liveness property (something good eventually happens) and therefore for infinite automata generally undecidable.

Definition 17. A "protocol" \mathcal{P} is a closed CBR product automaton characterized by the set of factor automata $\{\mathcal{A}_i\}$ and a set of Shannon Channels $\{c_j\}$. I call the factor automata "roles".

If not stated otherwise, I assume that a protocol is linear-executable.

Proposition 11. A protocol is linear-executable if its roles provide at most one character per output vector.

Proof. As each step of the protocol is achieved by only one component automaton (one role), the input and output components of the product automaton again differ from the empty character epsilon in at most one component.

For a product automaton, the properties of being well formed and being consistent are both not compositional as they are only properties of the protocol and not of its parts, the roles.

As was already mentioned in [44] the definition of "consistent" directly entails that a consistent protocol neither has deadlocks in the sense that there is a single reachable state without any continuation such that the acceptance condition holds nor livelocks, characterized by a set of periodically reached states without such a continuation.

5 Cooperating systems

Now we turn our attention to the systems themselves that cooperate, that is, take part in at least two different nondeterministic interactions in two different roles. I call such a system a process.

Definition 18. A system is called a "process" if it takes part in at least two different, well defined, consistent, and nondeterministic (protocol-)interactions in two different roles.

Definition 19. A process is called "linear-executable" if all its roles of all interactions it is involved in are linear-executable.

That is, a process is linear-executable, if it produces only one character per Shannon channel at a time.

Looking from an interaction perspective onto a process, as a system taking part in multiple nondeterministic interactions, it is an interesting question how to synthesize such a process from its roles. As described in [46], we thereby look for a second mechanism to couple roles, in this case to represent the causal relation between the transitions of the roles of a single system as an inner coupling determined by coordination rules, illustrated in Fig. 12.

5.1 Inner coupling: condition based restriction

The goal is to restrict the transition set of a weakly synchronized product automaton such that at least quasi-determinism of the formerly nondeterministic transition set is achieved in a sense that from each reachable state there is at most one transition for each input character and otherwise a single transition with the empty character as input, the collective acceptance condition can still be met, and the factor NIOAs (the roles) still can be regained by projection onto their interaction.

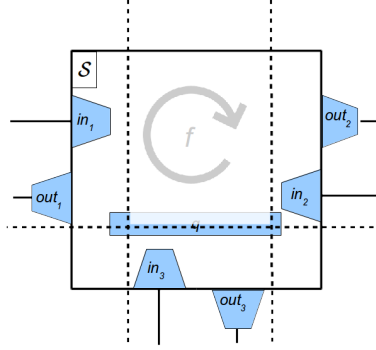


Fig. 12. A single system S coordinating multiple roles in different interactions. It shows the necessity to describe the inner coupling between the different roles, indicated by the circle arrow within the opaque area of the system.

Definition 20. An NIOA that has, from any reachable state, at most one transition for each input sign, and otherwise a single transition with the empty character as input is called "quasi-deterministic" (as is its transition relation).

Quasi-determinism means that, although not truly deterministic, there is no real choice anymore, such that an automated execution of all arising state transitions becomes possible.

Definition 21. Be \mathcal{B} an NIOA and $\pi = (\pi_Q, \pi_I, \pi_O) : Q_{\mathcal{B}}^\epsilon \times I_{\mathcal{B}}^\epsilon \times O_{\mathcal{B}}^\epsilon \rightarrow Q_{\mathcal{B}}^\epsilon \times I_{\mathcal{B}}^\epsilon \times O_{\mathcal{B}}^\epsilon$ a projection function. Then the projected automaton $\mathcal{A} = \pi(\mathcal{B})$ is given by $Q_{\mathcal{A}} = \pi_Q(Q_{\mathcal{B}})$, $I_{\mathcal{A}} = \pi_I(I_{\mathcal{B}})$, $O_{\mathcal{A}} = \pi_O(O_{\mathcal{B}})$, $q_{0\mathcal{A}} = \pi_Q(q_{0\mathcal{B}})$, $Acc_{\mathcal{A}} = \pi_Q(Acc_{\mathcal{B}})$, $\Delta_{\mathcal{A}} = \{(p', q', i', o') | (p, q, i, o) \in \Delta_{\mathcal{B}} \text{ and } p' = \pi_Q(p), q' = \pi_Q(q), i' = \pi_I(i), o' = \pi_O(o)\}$.

Definition 22. Be $\mathcal{T} = \bigotimes_{i=1}^n \mathcal{A}_i$ a weakly synchronized NIOA consisting of n nondeterministic roles \mathcal{A}_i and be (π_i) n projection functions, each projecting \mathcal{T} onto the Shannon channels of role \mathcal{A}_i . Then \mathcal{P} is a "coordinated automaton" of \mathcal{T} if $Q_{\mathcal{P}} = Q_{\mathcal{T}}$, $I_{\mathcal{P}} = I_{\mathcal{T}}$, $O_{\mathcal{P}} = O_{\mathcal{T}}$, $Acc_{\mathcal{P}} = Acc_{\mathcal{T}}$, and there exists a transition relation $\Delta_{\mathcal{P}} \subseteq \Delta_{\mathcal{T}}$ such that it is quasi deterministic, retains the collective acceptance condition, and retains all roles by projection onto their interactions, that is $\pi_i(\mathcal{P}) = \mathcal{A}_i$.

The transitions which are intended not to occur in the coordinated automaton can be described by a set of conditions - why I called this type of coupling "condition based coupling" in [46].

The fulfillment of all protocols can be seen as an invariant of the transition elimination procedure. Please not that (in contrast to what I have proposed in [46]), it could well be that the single roles still can be regained by projection, but the transition relation of the restricted product does not fulfill the collective acceptance condition any more.

An obvious and very interesting research question is, under which conditions such a coordinated automaton can be found.

Now that we have achieved a quasi-deterministic automaton by stating the coordinating conditions, we could either advance to a deterministic one and thereby creating a traditional process with an explicit system function. Or we could keep this quasi-deterministic automaton as it is and execute it in a modified execution environment.

Achieving full determinism by eliminating the remaining spontaneous transitions generally requires state value elimination, which implies a couple of serious consequences:

1. Eliminating state values possibly deprives all other factor automata from their chance to transit, possibly changing the behavior of the entire product automaton. That is, eliminating the spontaneous transitions changes the "atomicity of time steps".
2. As the acceptance components of the roles directly relate to the set of state values, they must be restated.
3. The projection relation between the restricted product automaton and its constituting roles gets lost.
4. If the product automaton was linear-executable before, we might have to aggregate several output characters in one step and thereby loose the property of linear-executability.

All these issues can be avoided by modifying the execution environment of a process by not only processing input characters as they become available (the

deterministic part), but also possible spontaneous transitions (the nondeterministic part).

This leads to an implicit process notion where rules related to the inner state of a system provide the internal glue to internally coupled roles together to become a process.

5.2 Process composition

How do processes compose? Can two systems, each taking part in at least two interactions, always be viewed as a single system, taking still part in at least two further interactions? Let us look first at a situation, where, for unlocked systems, no process results: There are three systems interacting as illustrated in Fig. 13 in a closed chain.

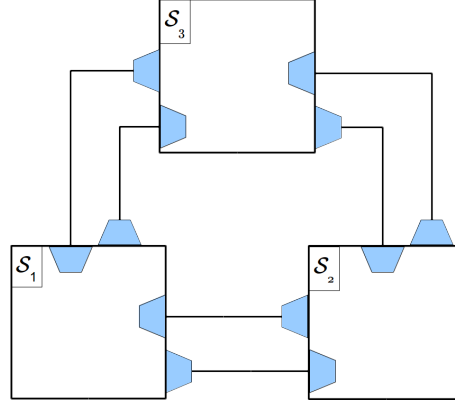


Fig. 13. Three systems, interacting mutually in a closed chain

If we look only at \mathcal{S}_1 and \mathcal{S}_2 , they seem to compose to a supersystem, having two new interactions. But both interactions refer to the same system \mathcal{S}_3 . If \mathcal{S}_3 has no other interfaces than the ones to \mathcal{S}_1 and \mathcal{S}_2 , then a closed chain similar to Fig. 5 results.

We see that an additional necessary condition beyond the remaining two interactions is, that they are directed towards two different systems.

Proposition 12. *Be \mathcal{S}_1 a linear-executable process with $n \geq 2$ roles $(\mathcal{A}_{1,i})$ and \mathcal{S}_2 a linear-executable process with $m \geq 2$ role $(\mathcal{A}_{2,j})$ which interact only through the consistent protocol $\mathcal{P} = (\{\mathcal{A}_{1,1}, \mathcal{A}_{2,1}\}, \{c_1, c_2\})$. Additionally, \mathcal{S}_1 and \mathcal{S}_2 each interacts with at least one additional linear-executable process \mathcal{T}_1 and \mathcal{T}_2 with $\mathcal{T}_1 \neq \mathcal{T}_2$ by consistent protocols. Then the protocol interaction creates a supersystem denoted by $\mathcal{S} = \mathcal{P}_{(\mathcal{A}_{1,1}, \mathcal{A}_{2,1})}(\mathcal{S}_1, \mathcal{S}_2)$ and this system is again a process.*

Proof. First we have to show that \mathcal{S} is indeed a system, then we have to show that this system still takes part in at least two interactions.

To show that \mathcal{S} is indeed a system, we have to construct a deterministic automaton from the deterministic automata of \mathcal{S}_1 and \mathcal{S}_2 . This is possible, since the well-definedness of \mathcal{P} guarantees that every interaction chain between \mathcal{S}_1 and \mathcal{S}_2 terminates. The linear-executability of both processes guarantees that any action of both, \mathcal{S}_1 or \mathcal{S}_2 , that was initiated by an interaction within \mathcal{P} terminates either with no output or with an output within another interaction.

According to the assumptions, the new system \mathcal{S} does interact with at least two other systems,

So, processes in this sense form unlimited interaction networks as was illustrated in Fig. 2. There is no such thing as an "end-to-end" process. What confines these interaction networks are the roles of the interaction protocols, that is incomplete projections of "the last interaction partner".

What happens if a process determines the behavior of another system entirely? As the following proposition shows, in the case of a consistent reciprocal system interaction, a system becomes a subsystem if its behavior is entirely determined by its interaction "partner". Such an interaction results in a recursive system where some output becomes the input of a next time step, until the chain of recursive interaction terminates

Proposition 13. *Let \mathcal{S} be a linear executable process with more than one consistent interactions and described by a DIOA \mathcal{D} . It especially interacts with another system \mathcal{U} described as a DIOA \mathcal{B} by the consistent linear-executable protocol $\mathcal{P}(\mathcal{A}, \mathcal{B})$, where \mathcal{A} is an NIOA describing only a projection of \mathcal{S} . Then \mathcal{S} and \mathcal{U} are subsystems of a larger system \mathcal{T} .*

Proof. The proof follows from the fact that the resulting automaton of two interacting linear executable DIOAs is again a linear executable DIOA and thereby represents a system - if in addition the interaction between the role of the process and the other complete system is a consistent and linear executable protocol. This implies that every interaction chain terminates. , one is determining the other in a consistent protocol interaction, is again deterministic - and thereby represents a system.

6 Related Work

There is a vast amount of existing literature on formal system engineering and modeling. Any extensive review would be an ambitious endeavor on its own. I restrict myself to a couple of other system models I came across while I was dealing with the subject. My focus is not so much on truth but on fit: Does a model fit to the phenomena it wants to represent in a way that it leads to adequate questions - and answers. I mainly look at how they represent the system function, how expressive their formal representation is and whether they reproduce the compositionality of the system function.

This investigation suggests that these three criteria do provide valid assessment criteria. To some extent, the expressiveness of the investigated system models does not suffice to represent the differences between parallel versus sequential composition or even recursion in a simple enough way or do not make a clear enough distinction between the deterministic versus nondeterministic interaction paradigms. Some lack the compositionality of their system function. And some other do not provide the formal means to distinguish between external and internal coupling.

6.1 Abstract data types and objects

According to the Dictionary of Algorithms and Data Structures, an abstract data type is "A set of data values and associated operations that are precisely specified independent of any particular implementation." Within the object oriented programming paradigm, this construct is in one way or another supplemented by polymorphism and inheritance.

According to definition 1 such an entity is a system, exposing its system function by its interface. The relation to partitioned I/O-automata was already expressed in Eq. 4 and 5. As the interaction between objects is expressed by a method call, this kind of interaction usually follows the pseudo-recursive case, illustrated in Fig. 10.

However, if the interaction between objects is based on untyped transport functionality then the semantics of the interaction is simply not expressed syntactically. If the interaction is based on mutual typed method calls, things become complex, as I have shown.

This explains that especially objects have their stronghold in the area of hierarchical system composition but fail to scale in the area of nondeterministic interactions.

6.2 Reactive systems

In [22] David Harel and Amir Pnueli distinguish between "transformational" and "reactive" systems. They define: "A transformational system accepts input, performs transformation on them and produces outputs" - which is consistent with the definition 1 of a system. They further define "Reactive systems ... are repeatedly prompted by the outside world and their role is to continuously respond to external inputs ... A reactive system, in general, does not compute or perform a function, but is supposed to maintain a certain ongoing relationship, so to speak, with its environment."

In my opinion it is false to conclude from the non-functional relation of the reactive systems to their interaction partners to their somehow non-functional functioning - or "mode of operation". Even reactive systems work in a step wise mode and at least for technical systems the simple fact holds that for being constructable, there has to be a function to be implemented - the system function. The main difference between "transformational" and "reactive" systems in the sense of David Harel and Amir Pnueli is not their functioning but - as they

correctly point out - their relation to their environment. For "transformational" systems their interfaces towards the interaction provides access to the full system function and thereby provides deterministic interactions. "transformational" systems are composed hierarchically by composition according to the definition of sub/super systems 8. By contrast, as I proposed already in [46], the interfaces of "reactive" systems provide access to at least two different projections of the systems, which I call "roles" and thereby only provide nondeterministic interactions.

6.3 The system model of Manfred Broy

Manfred Broy describes his component model in several articles (e.g. [10, 11, 12]). The behavior of a system is given by a set of "input" and "output" processes, whereby a process in his sense is a finite or infinite set of discrete events. The event concept is not entirely clear, as on the one hand, an event may represent (among others) a state change or some action, but on the other hand he states that an event represents a point in time and thus has no time duration. Also, it remains unclear, whether event sequences like a, ϵ, b and a, b are semantically equivalent or not.

A key concept in his theory is what he calls a *timed stream* of messages $m \in M$ which is an infinite sequence of finite sequences of messages. Each finite sequence $s \in M^*$ can be attributed to a time interval $t \in \mathbb{N}$ as $s(t)$. The set of timed streams is denoted as $(M^*)^\infty$. If we assume $M = \{0, 1, \dots, 9\}$, then M^* becomes the set of all natural numbers \mathbb{N} and $(M^*)^\infty$ is the set of all infinite sequences of arbitrary natural numbers. As can be seen from the example, the amount of information to be processed in one time interval is finite, but can be arbitrary large.

The messages are transported by typed channels $c \in C$. A *channel valuation* is a mapping which assigns a typed timed stream to channel. The set of all valuations to a given set of channels C is then defined as $\mathbf{C} = \{x : C \rightarrow (M^*)^\infty : \forall c \in C : x(c) \in (type(c)^*)^\infty\}$. Each channel valuation defines a communication history $x : C \rightarrow (\mathbb{N} \rightarrow M^*)$ for the channels in C .

The black box behavior or as Manfred Broy calls it, *semantic interface*, of a component is then described by a function f mapping the set of the histories of the input channels \mathbf{I} onto the power set of histories of the output channels $\wp(\mathbf{O})$ of the component: $f : \mathbf{I} \rightarrow \wp(\mathbf{O})$.

Manfred Broy then explicitly introduces the properties of causality as well as realizability in the sense that all output histories can indeed be computed.

Composition is defined by a single composition operator with feedback. Given two interface behaviors with disjoint set of output channels $O_1 \cap O_2 = \emptyset$: $F_1 : \mathbf{I}_1 \rightarrow \wp(\mathbf{O}_1)$, $F_2 : \mathbf{I}_2 \rightarrow \wp(\mathbf{O}_2)$. Then with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$, and $C = I_1 \cup I_2 \cup O_1 \cup O_2$ Manfred Broy defines the interface behavior of the composed interface $F = F_1 \otimes F_2 : \mathbf{I} \rightarrow \wp(\mathbf{O})$ by $F(x) = \{(y \in \mathbf{C}) | O : y | I = x | I \wedge y | O_1 \in F_1(y | I_1) \wedge y | O_2 \in F_2(y | I_2)\}$ where $x|Y$ means the restriction of the valuation x to the channels in Y .

According to Manfred Broy, the composition operator \otimes preserves realizability.

Due to the behavioral approach which focuses on observable behavior, the definition of black box behavior eschew internal states - but not for composed systems. The definition of a composed system allows for hidden channels which become nothing else than internal states in the sense of time dependent attributes of the composed system. Lets look at two simple systems $i = 1, 2$ each with two input channels $I_i = \{I_{i,1}, I_{i,2}\}$, two output channels $O_i = \{O_{i,1}, O_{i,2}\}$ and a deterministic system function $f_i = (f_{i,1}, f_{i,2}) = (I_{i,1} \wedge I_{i,2}, \neg(I_{i,1} \wedge I_{i,2}))$. The system composition is recursive with $O_{1,2} = I_{2,1}$ and $O_{2,1} = I_{1,2}$. By this equalities, we see that the channels do not just represent the channel names as Manfred Broy states, but the channels themselves. Otherwise the equality of two channel variables relating to two channels with two different names would not make much sense.

Following Manfred Broy's composition rule we then have $I = \{I_{1,1}, I_{2,2}\}$, $O = \{O_{1,1}, O_{2,2}\}$, $C = \{I_{1,1}, I_{1,2}, I_{2,1}, I_{2,2}, O_{1,1}, O_{1,2}, O_{2,1}, O_{2,2}\}$ and $F(x) = \{(y \in C) | O : y | I = x | I \wedge y | O_1 \in F_1(y | I_1) \wedge y | O_2 \in F_2(y | I_2)\}$.

This seems to be pretty simple and straight forward. So one is tempted to ask where the recursion has been hidden. The complexity is hidden in the fact, that without coupling, the streams $I_{1,2}$ and $I_{2,1}$ can take arbitrary values at each point in time, while in the coupled case, only their first value can be arbitrarily chosen and all other values depend in a complex way on these two initial values together with the complete history of input values of the remaining input streams $I_{1,1}$ and $I_{2,2}$.

So, the complexity of recursive system relations cannot be avoided. But recurring to the complex structure of timed streams, my impression is that Manfred Broy adds a substantial amount of unnecessary additional complexity whereas it is difficult for me to find the additional benefit.

6.4 The BIP component framework of Joseph Sifakis et al.

BIP stands for Behavior, Interaction, and Priority [5, 4, 6]. It is a general framework to describe reactive systems from certain parts with a set of rules. In my view, BIP is a way to describe computational systems in the sense of this article: a BIP-component provides states and a transition function. By its priority mechanism, together with a given port, there is supposed to be a unique transition to be selected.

An *atomic component* is a labeled finite transition system characterized by the tuple $(State, state_0, P, G, F, Trans)$. $State = S \times V$ gives the possible state values. S is called control state, $V = V_1 \times \dots \times V_n$ is called data state⁹. $state_0$ is the initial state. P is the set of ports, which are transition labels [or action names] used for synchronous state transitions of multiple components. G is a set of "guard" conditions operating on V , that is each $g \in G$ is a function $g : V \rightarrow$

⁹ the authors use V to denote the variables which represent the data states syntactically. Here the index i relates to the i -th data state [variable].

$\{true, false\}$. F is a set of functions operating on V , that is for each $f \in F$, $f : V \rightarrow V$. $Trans$ is the set of transitions with $trans = State \times P \times G \times F \times State$. A transition $t = (state, p, g_p, f_p, state')$ represents a step from state value (s, v) to (s', v') with $v' = f_p(v)$. It can be executed if some interaction including port p is offered by the execution environment and the guard $g_p(v) = true$.

Components are glued together by connectors defining interactions and by priorities. For simplicity we assume components with disjoint sets of names of ports, state values and transitions.

A *connector* specifies interactions between its components $\{C_1, \dots, C_n\}$. It is characterized by a tuple $(V, P, I, G, F, Trans)$. $V = V_1 \times \dots \times V_n$ is called data state¹⁰. P is the set of ports the connector relates to. $I \subseteq \mathcal{P}(P)$ is the set of feasible interactions, where each $i \in I$, $i \subseteq P$. G is a set of "guard" conditions operating on V , that is each $g \in G$ is a function $g : V \rightarrow \{true, false\}$ having access to the complete data state of all components. F is a set of functions operating on V , that is for each $f \in F$, $f : V \rightarrow V$, also having access to the complete data state of all components and providing "data exchange" between the components. $Trans$ is the set of transitions with $trans = V \times I \times G \times F \times V$. A transition $t = (v, i, g_i, f_i, v')$ represents a step from state value v to $v' = f_i(v)$. It can be executed if its interaction becomes feasible (that is, the respective components have reached a state such that each provides transitions labeled with the respective port) and the guard $g_p(v) = true$.

A *priority* selects among possible interactions. It is characterized by a tuple $(c, G, I, <)$. c is the component the priority belongs to, which composes $\{C_1, \dots, C_n\}$ with data state $V = V_1 \times \dots \times V_n$. G is a set of "guard" conditions operating on V . I is a set of interactions. $< \subseteq I \times I$ is a strict partial order relation providing the priority. When the condition holds and both interactions are enabled, only the higher one is possible.

A *compound component* is then given by its components, its connectors and its priorities. Neglecting priorities, it is finally characterized by a tuple $(State, P, I, G, F, Trans)$. $State = S \times V = S_1 \times \dots \times S_n \times V_1 \times \dots \times V_n$. $P = \bigcup P_i$, $I = \bigcup I_i$, $G = \bigcup G_i$, and $F = \bigcup F_i$ is the set of functions, operating on V . $Trans \subseteq State \times I \times G \times F \times State$ is the set of transitions where a transition is given by $t = (state, \alpha, g, f, state')$. α is a feasible interaction associated with a guard $g_\alpha \in G$ and a function $f_\alpha \in F$ such that there exists a subset $J \subseteq \{1 \dots n\}$ of atomic components with transitions $\{(state_j, p_j, g_j, f_j, state'_j)\}_{j \in J}$ and $\alpha = \{p_j\}_{j \in J}$. $g = \left(\bigwedge_{j \in J} g_j\right) \wedge g_\alpha$. $f = (f_1, \dots, f_n) \circ f_\alpha$. $v' = f(v)$. $s'(j) = s'_j$ if $j \in J$; otherwise $s'(j) = s_j$. That is, the states from which there are no transitions labeled with ports in α , remain unchanged.

Without priorities, a move $(s, v) \xrightarrow{\alpha} (s', v')$ is possible if there exists a transition $(state, \alpha, g, f, state')$, such that $g(v) = true$.

Most importantly, the functionality of BIP components is not compositional in the sense of definition 3. There is always the global function operating on all states of a composed component at once first. And only afterwards, the com-

¹⁰ Now the index i relates to the i -th component.

ponents' genuine functions are applied to the individual states, violating the encapsulation and thereby making this model of reactive systems complex. The purpose of this connector-function is probably to represent data exchange - but obviously it is not limited to reproduce states, but it is allowed to make arbitrary manipulations. So fundamentally, this model does not distinguish clearly between transport and processing of information.

6.5 The system model of Rajeev Alur

In his excellent book "Principles of Cyber-Physical Systems" [1] Rajeev Alur distinguishes between functional and reactive components in the tradition of David Harel and Amir Pnueli.

Focusing on the timing behavior, he develops several models of computation for reactive components. In the synchronous model all components execute in lock-step, similar to what I have called "clocked" systems. In the asynchronous model all processes execute at independent speeds, and there is an unspecified delay between the reception of inputs and the production of outputs by a process. In the timed model processes rely on a global physical time to achieve a loose form of synchronization.

Synchronous Reactive Components (SRC) A *synchronous reactive component (SRC)* C is defined by a tuple $C = (I, O, S, Init, React)$, mixing semantic elements like sets, states, and values with syntactic elements like variables and descriptions. I, O, S are sets of typed input, output and state variables, each variable taking values only from the sets Q_I, Q_O, Q_S . $Init$ is the description of the initialization defining the set $\llbracket Init \rrbracket \subseteq Q_S$ of initial states, and $React$ is a description of the reaction defining the transition relation $\llbracket React \rrbracket \subseteq Q_S \times Q_I \times Q_O \times Q_S$.

An SRC C is said to be *event triggered*, if a subset $J \subseteq I$ exist, where all the variables in J also can take the value \perp (meaning "absent"); every output variable is either latched (the value of the output variable is the updated value of a state variable) or can also be absent; and for a reaction with absent input, the non-latched output remains also absent.

The reaction $React = (L, \mathcal{A}, \succ)$ is given by a set L of task local variables and a set \mathcal{A} of tasks and a binary precedence relation $\succ \subseteq \mathcal{A} \times \mathcal{A}$ (The latter two represent a *task-graph*). Each task A has a read-set $R \subseteq I \cup S \cup O \cup L$, a write-set $W \subseteq O \cup S \cup L$, and an update description $Update$ with $\llbracket Update \rrbracket \subseteq Q_R \times Q_W$ such that (1) \succ is acyclic; (2) each output variable belongs to the write-set of exactly one task; (3) if an output or a local variable y belongs to the read-set of a task A , then there exists a task A' such that y is in the write-set of A' and $A' \succ^+ A$; and (4) if a state or a local variable x belongs to the write-set of a task A and also to either the read-set or write-set of a different task A' , then either $A \succ^+ A'$ or $A' \succ^+ A$.

As he says, his approach to ensuring well-behaved composition relies on the syntactic decomposition of the reaction description into tasks given by the designer. He introduce the notion of an interface for $I, O, \succ \subseteq I \times O$ of a component to ensure consistent composition. Two components C_1 and C_2 with I_i, O_i , and

\succ_i ($i = 1, 2$) are said to be *compatible* if (1) O_1 and O_2 are disjoint and (2) the relation $\succ = \succ_1 \cup \succ_2$ is acyclic. Actually, the task graph over the set of tasks of C_1 and C_2 obtained by retaining the precedence edges in the individual components and adding cross-component edges from a task A_1 of one component to a task A_2 of another component whenever A_1 writes a variable read by A_2 , is acyclic.

For composition, we assume that the input/output variables are named in a way to establishes the intended communication pattern and the internal variables are named to avoid naming conflicts. Let $C_i = (I_i, O_i, S_i, Init_i, React_i)$, $i = 1, 2$ be two compatible SCR. Suppose that $React_i = (L_i, \mathcal{A}_i, \succ_i)$. Then the *parallel composition* $C = C_1 || C_2$ is again an SCR such that $S = S_1 \cup S_2$, $O = O_1 \cup O_2$, $I = I_1 \cup I_2 \setminus O$, the initialization for a state variable x_i is given by $Init_i$, ($i = 1, 2$); the reaction description of C uses the local variables $L = L_1 \cup L_2$ and is given by the task graph such that (1) $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ and (2) $\succ = \succ_1 \cup \succ_2$ together with the task pairs (A_1, A_2) , such that A_1 and A_2 are tasks of different components with some variable occurring in both, the write-set of A_1 and the read-set of A_2 .

The commutativity of parallel composition of SCRs indicates that the formalism does not express the different composition schemata described in section 3. With the compatibility condition recursion is excluded.

Asynchronous processes In the asynchronous case, Rajev Alur talks about processes and channels instead of components and input/output variables.

An *asynchronous process* (AP) P is defined again by the tuple $P = (I, O, S, Init, React)$. However, the reaction is defined differently with input task A_x for every input channel $x \in I$, an output tasks A_y for every output channel y and a set of internal tasks \mathcal{A} as $React = (\{\mathcal{A}_x | x \in I\}, \{\mathcal{A}_y | y \in O\}, \mathcal{A})$. Each task consist of a guard condition *Guard* over S and an update rule *Update*. The update rule of each A_x has a read-set $S \cup \{x\}$ and a write-set S , each A_y has a read-set S and a write-set $S \cup \{y\}$, and each A 's read- and write-set is S .

The *parallel composition* $P_1 || P_2$ of two AP $P_i = (I_i, O_i, S_i, Init_i, React_i)$, ($i = 1, 2$) is again an AP if O_1 and O_2 are disjoint and $S = S_1 \cup S_2$, $O = O_1 \cup O_2$, $I = I_1 \cup I_2 \setminus O$, and $Init = Init_1; Init_2$. For each input channel $x \in I$, (1) if $x \notin I_2$, then the set of input tasks \mathcal{A}_x is \mathcal{A}_x^1 ; (2) if $x \notin I_1$, then the set of input tasks \mathcal{A}_x is \mathcal{A}_x^2 ; and (3) if $x \in I_1 \cap I_2$, then for each task $A_1 \in \mathcal{A}_x^1$ and $A_2 \in \mathcal{A}_x^2$, the set of input tasks \mathcal{A}_x contains the task described by $Guard_1 \wedge Guard_2 \rightarrow Update_1; Update_2$. For each output channel $y \in O$, (1) if $y \in O_1 \setminus I_2$, then the set of output tasks \mathcal{A}_y is \mathcal{A}_y^1 ; (2) if $y \in O_2 \setminus I_1$, then the set of output tasks \mathcal{A}_y is \mathcal{A}_y^2 ; (3) if $y \in O_1 \cap I_2$, then for each task $A_i \in \mathcal{A}_y^i$ ($i=1,2$) the set of output tasks \mathcal{A}_y contains the task described by $Guard_1 \wedge Guard_2 \rightarrow Update_1; Update_2$; and (4) if $y \in O_2 \cap I_1$, then for each task $A_i \in \mathcal{A}_y^i$ ($i=1,2$), the set of output tasks \mathcal{A}_y contains the task described by $Guard_2 \wedge Guard_1 \rightarrow Update_2; Update_1$; The set of internal tasks is $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$.

State machines However, to introduce more advanced matter like safety or liveness requirements, Rajeev Alur turns to state machines, very similar to the I/O-automata in this article.

Both, SCRs (p.66) and APs (p.141) have naturally associated transitions systems by declaring the input and output variables to be local variables. With a pair of states (s, t) he identifies $s \xrightarrow{i/o} t$ as the reaction for some input i and some output o for SCRs - which is exactly the notation of a transition of the DIOA. For APs he adds the undefined value \perp to the possible values of input and output variables in case they are not involved in an action - which is one of the essential ideas distinguishing NIOAs from DIOAs.

So, from an automata perspective, the difference between SRCs and APs is mapped onto the deterministic versus nondeterministic issue - very similar to the approach of this article. However, what is lacking is recursion in the deterministic case and most importantly the role concept and therewith the distinction between the inner and outer coupling, the essence of the process model of this article.

6.6 Algebraic approaches based on named actions

There are many algebraic approaches to describe processes which are based on named actions. Among others, these are Robin Milner's calculus of communicating systems (CCS) [40, 38], Charles A. R. Hoare's calculus of communicating sequential processes (CSP) [24] and Jan A. Bergstra's and Jan W. Klop's algebra of communicating processes (ACP) [7]. Also, Richard Mayr's [36] process rewrite systems can be subsumed her. For a recent overview, see the book of Jos C. M. Baeten, Twan Basten, and Michel Reniers [2].

They all have in common the view of a process as a structure producing events by actions and - this is the main point - are based on providing names for these events as their 'alphabet' they operate on. So, the semantics of these calculi is provided by transition systems where each transition has a label, the name of the action causing the state change. An interaction becomes the simultaneous execution of actions.

Although, it is generally possible to find a mapping from the i/o-pairs of the IO-automata used in this article to some set of unique transition labels, the coupling mechanism thereby gets lost. As described in section 4.2 the outer coupling was based on identical names of output and input characters: that the output of one transition is the input of another transition (Shannon channel). As described in (section 5.1), the basis for the inner coupling are the ϵ transitions, as these can be eliminated to represent the desired causal relation between the role-transitions.

As a result, an important difference between any approach with named actions and the presented approach with anonymous actions but named I/O-characters is their different support for describing composition behavior. As the names of the actions are arbitrary, they do not help to express the fact that some transitions may refer to the same action, but only from a different perspective/projection.

7 Discussion

This is a rather conceptual contribution where I combined the description of systems with the description of reciprocal system interactions in a network context. Basing the system notion on some identifiable functionality implies that the gestalt of systems becomes very dynamic.

Interactions between systems may create temporary super systems. A child that is hit by a car becomes aware of the center-of-mass system including the car only during the very impact. Someone speaking to me at close distance acquires control of my ear tympanum and thereby this part of me temporarily becomes a part of the speaker's system. Transferring this dynamics to computer science implies that any programming paradigm with a fixed linkage between data and functionality cannot fully represent this kind of system dynamics of the real world.

The presented system notion entails that the borders of systems are drawn by logic and sometimes practically restricted by physics. The borders between the internal "deterministic" parts of an engineered system and the parts that are coupled to it by nondeterministic interactions are drawn somewhat discretionary but are limited - from an engineering perspective - upon the required level of determinism. Effectively, determinism is fiction. In a recent study, Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber [52] measured the number of correctable errors of a DRAM-DIMM as nearly 4000 per year or 10 per day. Accordingly, the hardware manufacturer have to use error correction mechanism to bring this rate down below an acceptable limit. On the other hand, unreliable communication can make reaching consensus impossible, as the coordinated attack problem shows [21]. So, one effect of the increasingly faster and more reliable communication is to allow engineers to vastly extend the deterministic fiction to control even "remote" systems.

The presented approach shows that there is more to supersystem formation than mere interaction, as for example Robin Milner [39, 40] or Charles Hoare [24] still thought. Supersystem formation in the sense of this article becomes dependent on perspective and knowledge. It requires the presence of a well defined time successor function. Whether such a time successor function exists or not isn't always obvious and cannot be algorithmically decided in the general case.

Recursion makes the question where system borders begin and end, quite complex. As illustrated in Fig. 7 recursively interacting systems could be part of a loop- or a while-system, they could interact loosely coupled or only pseudo recursively - or they could do something thereof we must better be silent. In today's applications, unclear component relations are daily fare. Either there exist explicitly declared mutual functional dependencies or no declared dependency at all, as in the case of the broadly adopted web-programming paradigm AJAX - a security nightmare. With current practices, where operations represent only send or receive semantics and thereby not determining subsystem relations on the one hand and with protocol based interactions which in the deterministic case may determine subsystem relations, the questions where systems logically end is difficult to decide.

From a practical point of view, security mechanisms like encryption and signature could be viewed as tools to make system borders unequivocally visible. Based on asymmetric cryptographic procedures, a signature function uses the "private" key of a document issuer while a decryption function uses the "private" key of a document receiver. This notion of "privacy" relies on valid system borders and means that this kind of functionality is accessible only within a given system. If we look with this perspective on the current web clients, running in unknown environments (e.g. [33]), we see that it could be possible for the one who controls the environment to access the private client keys - invalidating the assumed system borders.

Well defined system borders are also a prerequisite for the "end-to-end" argument [49]. In their article, Jerome H. Saltzer, David P. Reed and David D. Clark stated that "choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer" and suggested as a guiding design principle for placement of function among the modules of a distributed system that "functions placed at low levels of a system may be redundant or of little value compared with the cost of providing them at that low level".

The demonstrated tight connection between our system and our function notion contributes to a better understanding, why approaches that rest mainly on the function notion struggle so much with modeling network-like interacting systems and the necessary "loose coupling". One example is functional programming [26], other examples are the many distributed object models like CORBA [56], Java RMI [57], COM/DCOM [48], SOA [53], etc.

Robert B. Glassman [19] defines interacting systems like cells, organisms or even societies as being "loosely coupled" if they have few states in common, or, to put it more technically, if the state space of the interaction is small against the state space the computed functionality is based on. However, within the object and service oriented communities, it is often stated that "loose coupling" of systems is created by abstraction (e.g. [29]). Though abstraction decouples from any specific implementation, it does not decouple from the particular implementation which is used during runtime and certainly not from the underlying logical mapping. Actually, this would imply to name a tire "loosely coupled" to a car, just because it can be changed easily. It seems to me that this kind of "loose coupling" is more concerned with low versus high effort to replace a subsystem.

The presented process model underlines that nondeterminism and therefore nondeterministic automata have an important role in software engineering (e.g. in contrast to Helmut Balzert [3], p.323). Nondeterminism provides the base for the role concept and thereby the differentiation between the roles' external and internal coupling. It avoids the strong centralization tendencies of the traditional approach to model processes as stepwise executable activities (see [45, 46, 15]). It explains, why execution environments which enforce the construction of deterministic processes also enforce a reduced flexibility on a software architecture level. Hence, the proposed process model signifies a new kind of reuse in software engineering, where processes can easily be configured from predefined roles by simply stating the rules for their internal coupling.

I would like to emphasize the break between the representation of systems by general DIOAs and processes, where linear executability becomes important to keep problems analytically tractable. It is the distinction between different interaction partners which makes it impossible to subsume several output states into a single vectorial output state and thereby subsume channels.

Another interesting distinction is the one between clocked and unclocked systems. While closed chains of unclocked systems are dead, as is illustrated in Fig. 13, this is not true for clocked systems. Actually, its enough to have a single clocked system in such a chain to enable activity. As "clocked" does not necessarily means to be triggered in regular, fixed intervals, or to work in sync with other systems, it could also be interpreted as the difference between systems with internally driven, truly spontaneous activity and systems which are purely passive.

It may also be interesting to transfer this insight to other scientific disciplines, like sociology, where both, the role concept (e.g. [20]) as well as the system concept (e.g. [42]) has already been very influential. Consequentially, one could speculate that this kind of rule based, loose inner coupling between roles to processes has its biological analogon within our central nervous system. On a more physiological level, one could speculate that the cerebellum with its important function in learning automated movements [35] is somehow a rule engine coordinating primitive movement patterns to more complex composites. Also the economic science of organizing organizations may benefit from the proposed process model, as at least some influential constructs like Lean [41] view processes as something static, inflexible, which has to be described as meticulous as possible to become amenable to engineered change. This could be extended to business software, whose task is to support the workflows in companies. The main difference between small and large companies is not so much their interface towards the outside, how they sell and buy, but rather the way and the amount of division of labor inside the company. So a business software supporting small as well as large companies alike might benefit enormously from structurally anchoring such a flexible role based process model in its software architecture.

The presented point of view on systems and their interactions also implies some important requirements on the completeness of component models. George T. Heineman and Bill Council ([23], p.7) define a software component as "*a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*" and a component model as a "*specific interaction and composition standard*".

A complete component model must at least support two general classes of components: The first class are components which are intended to behave non-deterministically in interactions. To be described adequately, these interfaces have to be protocol-like.

The second class are components which are intended to behave deterministically in interactions. This class can be further partitioned into components designated to work within a pipe with some splitting and merge-mechanism and

components designated to work as objects with lumped input and output, or as I called it "pseudo-recursively" in a component hierarchy.

In their overview, Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron [14] differentiate between "operation based" and "port based" interface support and thereby show that there are contemporary widely used, important component models which are not complete in my sense.

For the second class of components, an important requirement is to hide functional recursion and thereby to provide compositional behavior that is easy to understand. This requirement is automatically fulfilled if we stick to the proposed classification.

Let me conclude this article with the remark, that what computer scientists named "abstract data types" or "object" since the 1960s are in fact systems in the sense of this article. So, one could say that finally with the advent of cyber physical systems, it is time to change terminology and switch from "object" to "system" orientation, and - while keeping in mind the dynamics of the system notion - use the same terms for the same things as all the other engineering disciplines.

References

- [1] Alur, R.: Principles of Cyber-Physical Systems. MIT Press (2015)
- [2] Baeten, J.C., Basten, T., Reniers, M.A.: Process algebra: equational theories of communicating processes, vol. 50. Cambridge university press (2010)
- [3] Balzert, H.: Lehrbuch der Softwaretechnik. Spektrum Akademischer Verlag Heidelberg, Berlin, 2 edn. (2001)
- [4] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28, 41–48 (2011)
- [5] Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM), 2006. pp. 3–12. IEEE (2006)
- [6] Bensalem, S., Bozga, M., Quilbeuf, J., Sifakis, J.: Knowledge-based distributed conflict resolution for multiparty interactions and priorities. In: Formal Techniques for Distributed Systems, pp. 118–134. Springer (2012)
- [7] Bergstra, J.A., Klop, J.W.: A universal axiom system for process specification. CWI Quarterly 15, 3–23 (1987)
- [8] von Bertalanffy, L.: General System Theory: Foundations, Development, Applications. George Braziller, New York (1968)
- [9] Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)
- [10] Broy, M.: Mathematical system models as a basis of software engineering. In: van Leeuwen, J. (ed.) Computer Science Today, Lecture Notes in Computer Science, vol. 1000, pp. 292–306. Springer (1995)
- [11] Broy, M.: Relating time and causality in interactive distributed systems. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security, vol. 22. IOS Press (2009)

- [12] Broy, M.: A logical basis for component-oriented software and systems engineering. *Comput. J.* 53(10), 1758–1782 (2010), <http://dx.doi.org/10.1093/comjnl/bxq005>
- [13] Buede, D.M.: *The engineering design of systems: models and methods*, vol. 55. John Wiley & Sons (2011)
- [14] Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V.: A classification framework for software component models. *Software Engineering, IEEE Transactions on* 37(5), 593–615 (2011)
- [15] Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. *IEEE Trans. Software Eng.* 31(12), 1015–1027 (2005)
- [16] Farwer, B.: omega-automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) *Automata, Logics, and Infinite Games*. *Lecture Notes in Computer Science*, vol. 2500, pp. 3–20. Springer (2001)
- [17] Fielding, R.: *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine (2000)
- [18] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1 edn. (1995)
- [19] Glassman, R.B.: Persistence and loose coupling in living systems. *Behavioral Science* 18, 83–98 (1973)
- [20] Goffman, E.: *The presentation of self in everyday life* (dt.: *Wir alle spielen Theater*). University of Edinburgh Social Sciences Research Centre (1959)
- [21] Gray, J.: Notes on data base operating systems. In: *Operating Systems, An Advanced Course*. pp. 393–481. Springer-Verlag, London, UK (1978)
- [22] Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) *Logics and Models of Concurrent Systems*, pp. 477–498. Springer-Verlag, New York (1985)
- [23] Heineman, G.T., Councill, W.T. (eds.): *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
- [24] Hoare, C.: *Communicating Sequential Processes*. Prentice Hall (1985/2004), <http://www.usingcsp.com/cspbook.pdf>
- [25] Holzmann, G.J.: *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1991)
- [26] Hudak, P.: Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21(3), 359–411 (1989)
- [27] IEC 60050 International Electrotechnical Vocabulary (2001ff)
- [28] ISO/IEC 15288 System life cycle processes (2014)
- [29] Kaye, D.: *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 1 edn. (2003)
- [30] Kleene, S.: General recursive functions of natural numbers. *Mathematische Annalen* 112(5), 727–742 (1936)
- [31] Lau, K.K., Wang, Z.: Software component models. *IEEE Transactions on Software Engineering* 33(10), 709–724 (2007)
- [32] Lee, E.A.: *Cyber physical systems: Design challenges*. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley (Jan 2008)
- [33] Lekies, S., Stock, B., Johns, M.: 25 million flows later: Large-scale detection of dom-based xss. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. pp. 1193–1204. CCS '13, ACM, New York, NY, USA (2013), <http://doi.acm.org/10.1145/2508859.2516703>

- [34] Lynch, N.A., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. Tech. Rep. MIT/LCS/TR-387, MIT (1987)
- [35] Manto, M., Bower, J.M., Conforto, A.B., Delgado-García, J.M., da Guarda, S.N.F., Gerwig, M., Habas, C., Hagura, N., Ivry, R.B., Mariën, P., et al.: Consensus paper: roles of the cerebellum in motor control—the diversity of ideas on cerebellar involvement in movement. *The Cerebellum* 11(2), 457–487 (2012)
- [36] Mayr, R.: Process rewrite systems. *Information and Computation* 156, 264–286 (1999)
- [37] Mealy, G.H.: A method for synthesizing sequential circuits. *Bell System Technical Journal* 34(5), 1045–1079 (1955)
- [38] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (parts I and II). *Information and Computation* 100(1), 1–77 (1992)
- [39] Milner, R.: *A Calculus of Communicating Systems*. Springer, Berlin, Heidelberg, New York (1980)
- [40] Milner, R.: *Communication and Concurrency*. Prentice Hall (1989)
- [41] Ōno, T.: Toyota production system: beyond large-scale production. Productivity press (1988)
- [42] Parsons, T.: *The Social System*. New York: The Free Press (1951)
- [43] Reich, J.: Über Struktur oder das Verhältnis der Teile zum Ganzen. *Philosophia Naturalis* 38(1), 37–69 (2001)
- [44] Reich, J.: Finite system composition and interaction. In: Fähnrich, K.P., Franczyk, B. (eds.) 40. GI Jahrestagung (2). LNI, vol. 176, p. 603. GI (2010)
- [45] Reich, J.: Process synthesis from multiple interaction specifications. In: Heiß, H.U., Pepper, P., Schlingloff, H., Schneider, J. (eds.) 41. GI Jahrestagung. LNI, vol. 192, p. 309. GI (2011)
- [46] Reich, J.: Processes, roles and their interactions. In: Reich, J., Finkbeiner, B. (eds.) *Proceedings Second International Workshop on Interactions, Games and Protocols, IWIGP 2012, Tallinn, Estonia, 25th March 2012*. EPTCS, vol. 78, pp. 24–38 (2012)
- [47] Reich, J.: Eine semantische Klassifikation von Systeminteraktionen. In: Cunningham, D., Hofstedt, P., Meer, K., Schmitt, I. (eds.) *INFORMATIK 2015*. pp. 1545–1559. *Lecture Notes in Informatics (LNI)*, Gesellschaft für Informatik, Bonn (2015)
- [48] Rogerson, D.: *Inside COM*. Microsoft Press (1998)
- [49] Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. *ACM Transactions on Computer Systems*, Vol. 2, No. 4, November 1984, Pages 277–288. 2(4), 277–288 (1984)
- [50] Scantlebury, R.A., Bartlett, K.A.: A protocol for use in the NPL data communications network (1967), technical Memorandum
- [51] Schlager, K.J.: *Systems Engineering - Key to Modern Development*. IRE Transactions on Engineering Management pp. 64–66 (1956)
- [52] Schroeder, B., Pinheiro, E., Weber, W.D.: Dram errors in the wild: a large-scale field study. In: *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. pp. 193–204. ACM, New York, NY, USA (2009)
- [53] Schulte, R.W., Natis, Y.V.: "Service-Oriented" Architectures, Part 1 and 2. SSA Research Notes SPA-401-068, -069, Gartner Group (1996)
- [54] Shannon, C.E.: A mathematical theory of information. *Bell System Technical Journal* 27, 379–423, 623–656 (1948)
- [55] Sifakis, J.: A vision for computer science - the system perspective. *Central European Journal of Computer Science* 1(1), 1008–116 (2011)

- [56] Vinoski, S.: CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communication* 35(2), 46 – 55 (1997)
- [57] Wollrath, A., Riggs, R., Waldo, J.: A distributed object model for the javatm system. In: COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS). pp. 17–17. USENIX Association, Berkeley, CA, USA (1996)
- [58] Zimmermann, H.: OSI reference model - The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications* COM-28(4), 425–432 (1980)